Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

Ingrid Maria Sundfør

Flexibility in Distribution Systems through PyDSAL

Master's thesis in Energy and Environmental Engingeering,
Electrical Energy Engineering
Supervisor: Prof. Olav Bjarte Fosso

November 2022

NTNU
Norwegian University of
Science and Technology

Ingrid Maria Sundfør

Flexibility in Distribution Systems through PyDSAL

Master's thesis in Energy and Environmental Engingeering, Electrical Energy Engineering
Supervisor: Prof. Olav Bjarte Fosso
November 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

NTNU
Norwegian University of
Science and Technology

Sammendrag

Distribuert produksjon, lokal lagring og elektrifisering av transport og industrielle prosesser gir mange nye utfordringer for strømforsyning. Nye forbruksprofiler kombinert med lokal produksjon, der tilgjengelige ressurser viser signifikant variasjon, utfordrer distribusjonssystemet. Denne masteroppgaven kommenterer enkelte utfordringer ved å anvende et eksisterende verktøy (en prototyp) på et referanse-system klargjort av FME CINELDI. Hensikten med dette referanse-systemet er å ha et vel definert tilfelle av et nett representativt for en rekke norske distribusjonssystemer.

Analysene er utført på test-systemet CINELDI 124 busser for å finne dets optimale operasjon i ulike tilstander ved hjelp av det objekt-orienterte verktøyet PyDSAL (Python Distribusjons System Analyse Bibliotek). Algoritmen Framover-Baklengs Sveip brukes som verktøyets motor. Verktøyet har utvidet funksjonalitet og et skall (Algorithm B.2) er blitt utviklet for å tilrettelegge studiene. Spenningskontroll med tillegg/svinn av spenning er benyttet. Prinsippene og strategiene utviklet i denne masteroppgaven er anvendbare på andre distribusjonssystemer med lignende struktur og karakteristikker.

PyDSALs utbytte, bestående av profiler av spenning, kraftflyt og sensitiviteter for spenning og tap på grunn av endringer i aktive og reaktive injeksjoner, er nyttige for operasjonsavgjørelser. Programvarens evne til å løse alternative topologi-tilfeller ved å dele nettet og forsyne dets delnett med sikkerhetskilder og -koblinger, muliggjør nyttig analysering av alternative strategier for å forbedre forsyningssikkerheten. Konseptet kan innbefatte både mikrogrid operasjon og hovednett-tilkoblet tilstand. Resultatene er vist grafisk og diskutert i detalj i rapporten.

Se Appendix B for denne masteroppgavens versjon av programvaren.

Abstract

Distributed generation, local storage and electrification of transport and industrial processes give many new challenges for the distribution of electric energy. New consumption profiles combined with local generation where the available resources will show significant variation will give challenges for the distribution system. This thesis will address a number of these challenges by using an existing prototype tool on a reference system prepared by FME CINELDI. The purpose of this reference system is to have a well-defined case of a grid representative for a number of Norwegian distribution systems.

The analyses are done on the CINELDI 124 bus test system to find its optimal operation under different conditions using the object oriented tool PyDSAL (Python Distribution System Analysis Library). The Forward-Backward Sweep algorithm is used as the tool's engine. The tool has got extended functionality and a shell (Algorithm B.2) has been developed to facilitate the studies. Voltage control with a droop voltage approach is applied. The principles and strategies developed in this thesis will be applicable to other distribution systems with similar structure and characteristics.

PyDSAL's outputs, which include profiles of voltage, line flow and sensitivities for voltage and loss due to changes in active and reactive injections, are useful for operational decisions. Its ability to solve alternative topology cases by splitting the grid and supplying sub-grids from backup sources or backup connections, makes the tool useful to investigate alternative strategies to improve the security of supply. The concept may involve both microgrid operation and in grid connected mode. Results are depicted graphically and discussed in detail in this report.

See Appendix B for this thesis' version of the software.

Preface

This master thesis was written during the fall of 2021 until the fall of 2022. The host institution was the Norwegian University of Science and Technology (NTNU), Department of Electrical Power Engineering. The thesis is the last part of a five-year Master programme in Energy and Environmental Engineering, corresponding to 30 ECTS credits.

The objective of this thesis was suggested by my supervisor Professor Olav Bjarte Fosso. The topic is connected to SINTEF Energy Research's FME CINELDI and the studies conducted there concerning flexibility in system planning and operation. This thesis focuses on tool and strategy development to address system problems, such as a system split or supply from alternative feeders. In effect, Prof. Fosso's shell for PyDSAL is replaced by the shell created in this thesis (Algorithm B.2). Click here to arrive at the previous version of PyDSAL's shell. Thus the tool was expanded, enabling the flexibility in system planning/operation latent in PyDSAL.

I am grateful for the opportunity to dig deeper into this problem and be able to see the beauty and the ingenuity behind the solution.

Especially, I would like to thank both my supervisor and my brother for always believing in me and helping me along, I could not have completed my master's degree without them.

Ingrid Maria Sundfør, Haugesund, 12th November 2022

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

Table 0.1: Abbreviations

Abbreviation	Explanation
CINELDI	Centre for Intelligent Electricity Distribution - to empower the future Smart Grid.
FBS	Forward Backward Sweep algorithm (Section 1.3).
FME	The Centres for Environment-friendly Energy Research. Norwegian: Forskningssentrene for miljøvennlig energi.
FME CINELDI	”The scheme of the Centres for Environment-friendly Energy Research (FME) seeks to develop expertise and promote innovation through focus on long-term research in selected areas of environment-friendly energy. There are today 10 centres within renewable energy, energy efficiency, social sciences and CO2-management. The research activity is carried out in close cooperation between prominent research communities and users. The centres will operate for eight years (2016 – 2024).” [FME]
IEEE	Institute of Electrical and Electronics Engineers.
PyDSAL	Python Distribution System Analysis Library (Section 1.2).
SINTEF	The Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology. Norwegian: Stiftelsen for industriell og teknisk forskning.

Terminology

Table 0.2: Terminology, mostly paragraphed from a dictionary [Cop19]

Term	Explanation
active	Workable, capable of producing the desired effect or result; feasible.
algorithm	A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.
bus	Node; a point in a network or diagram at which lines or pathways intersect or branch.
case	An instance of a particular situation.
class object	A class is a preset format for implementing objects. Consequently, any variables, functions and methods implemented within a class are members of it. An object is a data construct that provides a description of anything known to a computer (such as a piece of code) and defines its method of operation.
configuration	An arrangement of parts or elements in a particular form, figure, or combination.
configure	Arrange or order a topology or an element of it so as to fit it for a designated task.
feeder	A distribution point that supplies the grid.
function	A set of instructions designed to perform a frequently used operation within a program, tailored to a specific task. Writing a function's name rather than its code in a script, shortens the script, dividing the responsibility of an algorithm's sequences to functions.
injection	An injection corresponds to power either draining (positive injection) from or filling (negative injection) a node. Correspondingly, a node is either a ”sinkhole” or a ”fountain”.
iteration	Repetition of a mathematical or computational procedure applied to the result of a previous application, typically as a means of obtaining successively closer approximations to the solution of a problem.
laws	Rules defining correct procedure or behaviour in an algorithm.
load flow	A numerical analysis of an electrified grid.
loop	A programmed sequence of instructions that is repeated until or while a particular condition is satisfied.
microgrid	A small network of electricity users with a local source of supply that is usually attached to a centralized national grid but is able to function independently.
node	A point in a network or diagram at which lines or pathways intersect or branch.
object-oriented	(of a programming language) Using a methodology which enables a system to be modelled as a set of objects which can be controlled and manipulated in a modular manner.
outage	A period when a power supply or other service is not available or when equipment is closed down.
output	The amount of something produced by an algorithm.
process	Operate on data by means of a program.
reactive	Directionless. Acting in response to a stimulus rather than creating or controlling it. Noisy.
script	An automated series of instructions carried out in a specific order.
sensitivity	A minor increase or decrease in the magnitude of a property (e.g. line flow, bus voltage or bus load) observed in passing from one node to another. A sensitivity can also be the rate of such a change, or the minor quantity of the change.
shell	A program which provides an interface between the user and the operating system.
simulation	The production of a computer model of a snap-shot of an electrified grid, especially for the purpose of study.
system	A set of things working together as parts of a mechanism or an interconnecting network; a complex whole.
topology	The way in which constituent parts are interrelated or arranged.

1 Introduction

1.1 Motivation

Distributed generation, local storage and electrification of transport and industrial processes create many new challenges for the distribution of electric energy. New consumption profiles combined with local generation where the available resources will show significant variation will create challenges for the distribution system. Most systems will experience lack of transfer capacity and problems with the quality of supply at least in periods. Though investment in new infrastructure may be necessary, it is important to be able to use the available capacity optimally. This will in most cases be to operate closer to the physical limits of the equipment while using load-shifting, topology changes, supply parts of the system from backup feeders, local storages, costumers willingness to change consumption profiles to reduce consumption as well as vehicle to grid (V2G). See Section 3 for more details. These system situations have in this thesis been implemented in the Python language as described generally in Section 2 and specifically in Section 4. As many of the loads and local generations as well as storage devices will be interfaced to the grid using Voltage Source Converters (VSC), this will provide an opportunity for costumers to actively contribute to the system services as voltage control and active reserves.

In such a new dynamic environment, it is important to adapt to the actual situation and quickly find a solution to the upcoming challenges. This will involve alternative topologies where the sub-systems are supplied from alternative feeders and some part of the system may temporally be operated as microgrids.

The major advantage of using a microgrid concept is that it allows for the use of locally produced power and the stored energy in the system to supply important loads over a longer period in an isolated mode, where the alternative would be to shut down the loads until the sub-system could be operated in connected mode again.

It is the locally produced energy and the ability to store energy that make such solutions feasible. This new dynamic world needs tools to quickly identify alternative solutions and configurations.

The core of this work is to further develop an existing prototype to make it appropriate for analysing a high number of alternative solutions and to provide the user with decision support tools. The studies will be connected to the CINELDI 124 bus reference system. This is a system prepared as a reference system representative for many distribution systems in Norway. It is based on a real system but anonymized to make it possible to conduct studies and publish the results without identifying the actual grid.

Existing tools to simulate microgrid systems have limitations on flexibility. Thus an open source tool to study system performance for different topologies is of interest, as it enables planning and operation of distribution and microgrid systems. More information about the tool can be found in Section 1.2.

This thesis' objective is to further develop the existing open source code, so that an optimal operation (while fulfilling local voltage and flow constraints) can be simulated and results may be used for decision support.

1.2 PyDSAL

In 2020 Prof. Fosso developed and published the open-source software PyDSAL (Python Distribution System Analysis Library) on Github [https://github.com/obfosso/PyDSAL], with available for download:

- A zip-file.
- A spreadsheet for a IEEE 66 bus test system.

The zip-file contains the version of the scripts listed in Table 1.1. Currently, the code developed is for radial systems, providing many sensitivities (Section 1.4), benefitting decision support. Further developing that code resulted in this thesis' main contribution Algorithm B.2. See Section 2 for more details on this thesis' contributions to the tool. The spreadsheet contains grid specification and parameters as generally described in Section 1.5.

Implemented in the Python language, the tool contains a shell, spreadsheets, class objects, laws and functions. Distribution load flow simulations are executed by the shell. A given case determines which grid relevant data is added to or excluded from a topology, followed by the simulation of the grid's electrification. The shell acts like a resettable game, where for each round, parts are added or removed before the game is ready to play that round. In other words, analogous to an actual clockwork mechanism:

- The shell drives the wheels in the clockwork.
- The laws determine the dimensions of the wheels.
- The functions are the wheels.
- The class objects are the wheels' axels.
- The grid parameters listed in the spreadsheet are the slings connecting the wheels.

If any of these parts are missing, the clockwork will not tick. A deep dive into the workings of the source code was done to master and further develop it. The shell is described in detail, whereas other parts of the code are not, as they go beyond the scope of this thesis. The shell is implemented to progress three main stages:

1. configure a distribution grid.
2. simulate a snap shot of power flowing in said grid.
3. display simulation outputs.

It is the second stage that is the actual simulation, illustrated by a green circle "Run simulation" in the flow chart in Figure 2.2. Delving into the workings of a simulation, the snap-shot electrification of the grid is executed in Algorithm B.2 by the function $DistLF$. Its name is abbreviated from "distribution load flow". $DistLF$ navigates the grid by the algorithm Forward Backward Sweep (FBS, Section 1.3). Meaning, an iteration of $DistLF$ calls in succession two functions:

1. $accload$

   accumulates every node's parameters of load and loss, led by a backward sweep.
2. $UpdateVolt$

   updates every node's parameters of voltage and sensitivities, led by a forward sweep.

If the simulation's outputs are found to be within acceptable limits, no reiterations are required, and the simulation is completed.

As of now the software performs analysis on a single-line system, although the real-life grid is a three-line system. Previous versions of the tool analysed a IEEE 33 bus test system, then a IEEE 66 bus test system. This thesis' version of PyDSAL, the scripts listed in Table 1.2, appended in Appendix B, is tailored to a CINELDI 124 bus test system (single-line diagram in Figure 1.3) and the simulation scenarios (Section 3). By continually increasing the complexity of the grid under analysis, the intention is to expand the object-oriented software from a tailored to a general and easy-to-use tool.

Table 1.1: An overview of the previous version of PyDSAL

Scripts	Purpose
`DistLoadFlow.py`	Laws, functions and shell
`DistribObjects.py`	Class objects
`MenuFunctions.py`	Selector of spreadsheets
`BuildSystem.py`	Reader of the selected spreadsheet

Table 1.2: An overview of this thesis' version of PyDSAL

Scripts	References	Purpose
`concept.py`	Algorithm B.2, Section 2.1 and a flow chart in Figure 2.2	Shell
`DistLoadFlow-vIngrid.py`	Algorithm B.3, Section 2.2	Laws and functions
`DistribObjects-vIngrid.py`	Algorithm B.4, Sections 4.3, 4.4, and 4.5	Class objects
`MenuFunctions-vIngrid.py`	Algorithm B.5	Selector of spreadsheets
`BuildSystem-vIngrid.py`	Algorithm B.6	Reader of the selected spreadsheet

Previously, the shell, laws and functions were all in one script, but were split into the two scripts Algorithm B.2 and Algorithm B.3, singling out the tool's shell as a stand-alone entity. Algorithm B.2 (flow chart in Figure 2.2) was the main work of this thesis, based on the previous shell located at the end of Algorithm B.3. Click here to arrive at the previous version of the shell. Mainly the tool's shell and functions have been further developed (Section 2), writing a new type of shell and altering the outputs' layout. Minor changes concerned the implementation of voltage dependent loads are detailed in Section 1.6.3. Also, a new code line was added to three class objects, addressed in Sections 4.3, 4.4 and 4.5.

1.3 Forward-Backward Sweep and a simulation's outputs

Forward-Backward Sweep (FBS) is PyDSAL's engine, enabling the simulation of a distribution load flow. As the name suggests, the algorithm procures a visit to every node in the grid in one sweep, before sweeping back in the opposite direction, thus revisiting every node. In both sweeps, every visited node is evaluated, and its electrical parameters calculated. The derivation of these parameters' equations are found in [Fos20] and [Haq95].

In effect, FBS consists of two of Algorithm B.3's functions, already commented on in Section 1.2:

1. $accload$

   constitutes a backward sweep, illustrated with a flow chart in Figure 1.1.

2. $UpdateVolt$

   constitutes a forward sweep, illustrated with a flow chart in Figure 1.2.

An iteration of these two functions in succession produces a simulation of an electrified grid's snapshot. Both functions require two lists, described in-depth in Section 2.1:

- BusList

  is the chronological list of the grid's buses, containing their electrical and topological parameters.

- TopologyList

  is the grid's tree structure, mirroring the single-line diagram in Figure 1.3.

Guided by the trail provided by TopologyList, both functions visit one bus at a time, steadily updating BusList. This sweeping procedure can be imagined as a light moving from node to node in the tree structure. When a node is visited, it is lit up while the others remain in darkness. Meaning that only this node is investigated now. As a sweep moves through the tree structure, the light jumps from branch to branch moving along the tree, until all buses have been evaluated.

In summary, a simulation is performed when in succession:

1. $accload$

   reverses TopologyList, updating BusList.

2. $UpdateVolt$

   trails TopologyList, updating BusList.

Thus FBS comprises these two functions, due to the direction they sweep TopologyList with.

Whenever a joint to a side branch is reached, FBS traverses it depending on the type of sweep. If it's a backward sweep, $accload$ jumps to the side branch's last node, navigating itself back to the main branch. If it's a forward sweep, $UpdateVolt$ visits the side branch's first node, trailing out the side branch, until it jumps back to the main branch.

During a backward sweep (see flow chart in Figure 1.1), every visited node's parameters of load and loss are calculated, using the status of previously processed nodes as input parameters. Comparing the tree structure with a river split into several channels, which themselves split into several channels etc., the accumulation sequence can be seen as an accumulation of water quantities, starting with visiting the node furthest downstream, which is the last node listed in TopologyList. The respective accumulated load and loss at a given bus is registered for further use in the upcoming forward sweep.

A completed backward sweep is followed by initiating a forward sweep (see flow chart in Figure 1.2). Its procedure starts with visiting the node furthest upstream, which is the first node listed in

TopologyList, working its way to the last node. By default, the first node's voltage is set to 1.0 pu, which is the grid's feeding point, setting the system's voltage reference. Moving forward through the grid, the incremental node's parameters of voltage and sensitivities are calculated, until the last node is reached. Comparing the tree structure with a river as before, *UpdateVolt* sees a node as an intersection of the river. The sequence of bus sensitivity calculations can thus be seen as a forecast of water quantities at the next appointed intersection based on the upstream intersection's quantities. A change upstream affects every downstream channel.

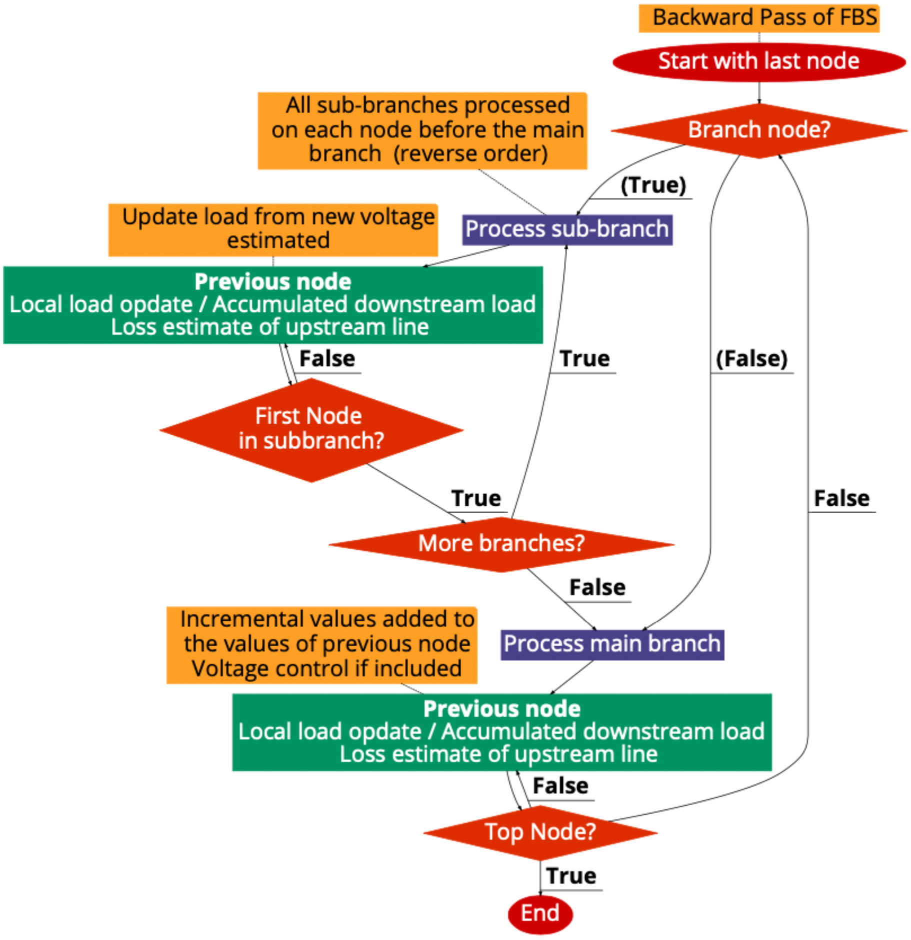


Figure 1.1: Flow chart of the steps in a backward sweep [Fos20]

Thus an iteration of a distribution load flow simulation is a backward sweep followed by a forward sweep. FBS's main principle is to estimate the loads and losses, sweeping them backwards, and to then calculate voltages and sensitivities, sweeping them forwards. If the resulting bus voltages converged, this imitation of an electrified grid at a moment in time is sufficient for analysis. Otherwise, another iteration will be performed. Then the voltages calculated from the previous iteration have overwritten the default setting of only 1.0 pu voltages in the grid, thus *accload*'s estimation of voltage dependent loads (Section 1.6.3) may differ now due to its updated voltage input. If the end of the loop of iterations is reached without a converged simulation solution, the loop is exited, and the end simulation is not valid.

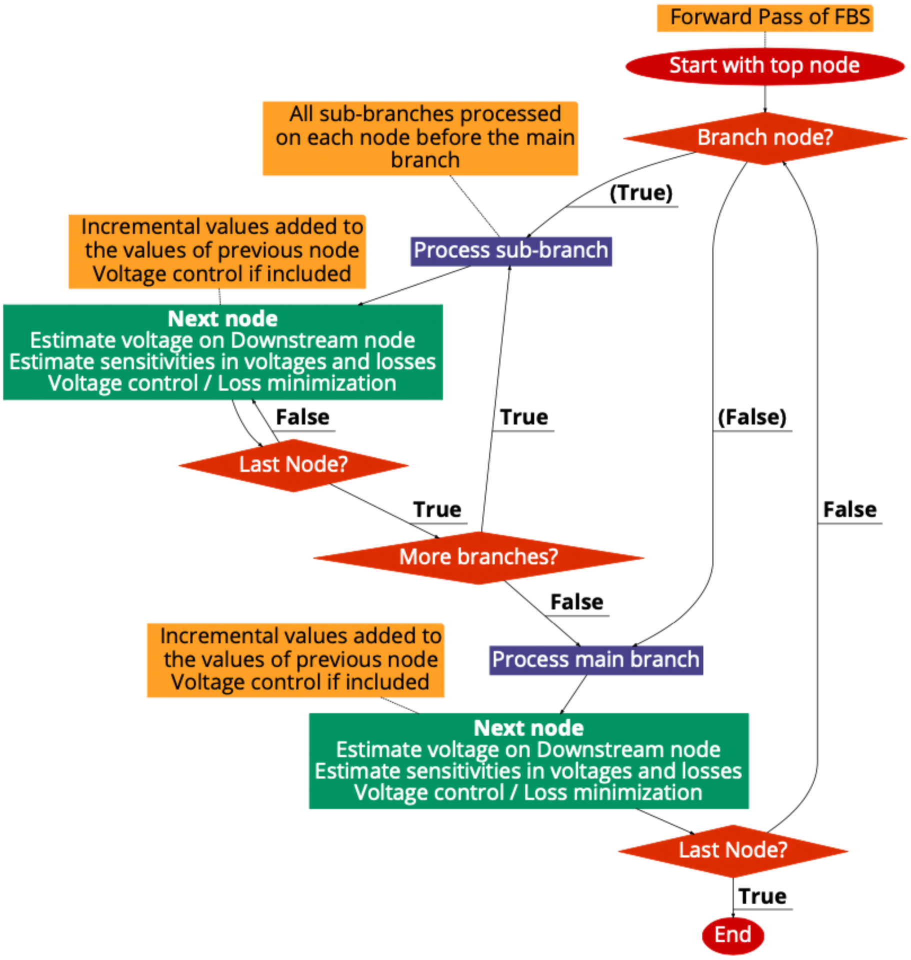


Figure 1.2: Flow chart of the steps in a forward sweep [Fos20]

1.4 A simulation's sensitivities and power loss minimization

PyDSAL's simulation outputs are extensive, as commented on in Section 5 and seen in Table 5.35, but the sensitivity outputs are not displayed in this report, downsizing this report.

During a simulation, the tool visits every node in the grid, calculating sensitivities as well as other node parameters. Every line is also visited, but no sensitivities concerning a line are calculated. The sensitivities all concern a bus injection's update. Thus they are meant to be utilized in the shell's injection-loop (Section 2.1), where optional bus power change is applied, or in PyDSAL's estimation of any added voltage dependent loads (Section 1.6.3).

The sensitivities may be used for decision support if the user needs to change for example the voltage profile or wants to minimize the losses by changing voltage set points on voltage controlling devices. It's up to the user to chart the recommended power corrections, and update the grid's load profile. Since none of this thesis' tasks covered loss minimization, no attempts were made to utilize the sensitivities.

See Table 1.3 for the three types of sensitivities PyDSAL offers, concerning a bus.

Table 1.3: A simulation's sensitivities categorized, concerning a bus

Type	Explanation
Type 1	The partial derivative of its voltage or power loss, with respect to its load: Quantifying its voltage and power loss change, with respect to its consumption/supply.
Type 2	The second partial derivative of its active power loss, with respect to its load: Quantifying how fast its active power loss changes, with respect to its consumption/supply.
Type 3	The correction of its load, with respect to its load: Quantifying the bus power change needed to adjust its injection, with respect to its consumption/supply.

Demonstrating a sensitivity evaluation, only the third sensitivity type is explained in further detail, downsizing this report. As shown in Equation 1 [Fos20], the correction of bus b's reactive power injection Q_b is calculated. It's the partial derivative of its power loss P_b^{Loss}, divided by the second partial derivative of said loss, all with respect to its double marked reactive power injection $Q_b^{"}$. The double marking indicates that the charge located over the bus's shunt is included in the bus's total reactive power injection.

$$\Delta Q_b = \left.\frac{\partial P^{Loss}}{\partial Q^{"}}\right|_b \left(\left.\frac{\partial^2 P^{Loss}}{\partial Q^{"2}}\right|_b \right)^{-1} \quad \text{at bus } b \tag{1}$$

The correction is a sensitivity, though defined as a fraction of two sensitivities: The denominator is the numerator's rate of change. In fact, type 3 equals a type 1 divided by a type 2.

If $Q_b^{"} \uparrow$, then $P_b^{Loss} \uparrow$, forcing $\Delta Q_b \uparrow$. Thus Equation 1 quantifies the reactive bus power injection needed to minimalize a bus's active power loss. Minimum bus active power loss is attained when the numerator is zero.

1.5 Grid specification and input parameters

The topology, specification and parameters were provided as part of the thesis assignment, and the objective was to further improve Prof. Fosso's work on analysing the CINELDI 124 (Figure 1.3) bus test system.

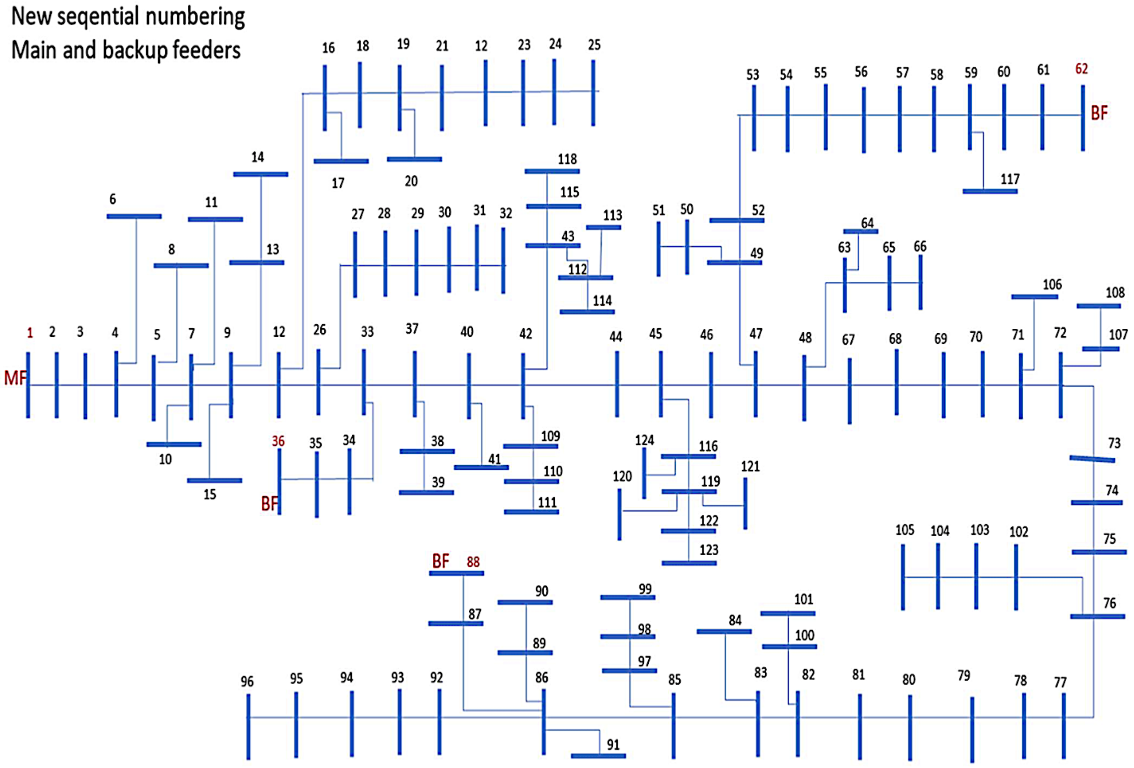


Figure 1.3: CINELDI 124 single-line diagram
Alternative feeders in red writing

The CINELDI 124 bus test system (from now on referred to as "the grid" for simplicity) is a 22 kV radial grid with 124 nodes. What every node encompasses is not made known, although the grid is based on an anonymous real-life grid. It is implemented with a standard load profile (snap-shot for one time interval) and as a stand-alone grid, although it in reality has a complex load profile and is supplied by a higher voltage levelled grid. The higher level grid is not modelled here so the feeding node is assumed as a stiff voltage (fixed voltage level and zero angle).

The grid serves as a distribution network. A transmission network is where a distribution network receives its power, from alternative supply points. The grid's alternative feeders are all connected to the same transmission network, marked with red writing in Figure 1.3: main feeder B1, backup feeders B36, B62 and B88.

The grid is described by the following electric input parameters at a moment in time:

- Bus voltage magnitudes and angles
- Active and reactive bus loads
- Line resistances and reactances
- Line flow limits

1.5.1 The main branch

For the numbering of the nodes it is chosen to try to keep the nodes close to each other in the same number range. This is not a requirement but convenient to quickly get an overview of the location of a bus. A main branch is a grid's highest priority load trail, thus transmits the grid's main flow. Closer inspection reveals that the grid's main branch consists of the buses B1-B88 when main feeder B1 is the node furthest upstream, feeding the grid. With four alternative feeders, with their four dispersed locations, the grid's node furthest upstream is different for every alternative. Thus the grid has for every feeder change a different main branch, as the flow branches out from the feeder. This is explained in Section 4.2.

1.5.2 Per unit measurement

The system is represented in pu-values. This means that this report's quantities are in accord with the reference values $V_{ref} = 22$ kV and $S_{ref} = 10$ MW. For the reader of this report, the dimension of S_{ref} is set to MW for simplicity, when it in fact is MVA, thus downsizing this report.

- A voltage of 1.0 in pu is then 22 kV line-to-line.
- A line flow or load of 0.1 pu is 1 MW.

1.6 Loads

The cases to be studied have local generation, storage and voltage control options. The standard load profile is modelled as a constant-power load. The grid's batteries are the local storage options, which also function as local generation during discharging. Voltage control options come into effect with voltage dependent loads present in the grid, e.g. batteries or EVs are connected to the grid.

An implemented positive bus injection, e.g. when a household is cooking dinner on an electric stove, corresponds to power being drained from the "household" bus. Comparably, an implemented negative bus injection corresponds to power being supplied to the system, e.g. when a storage battery supplies the grid at the "storage battery" bus.

1.6.1 Standard loads

The system is provided with a load profile based on the original system loads but scaled to enable a load increase or decrease. This is needed to demonstrate some of the challenges imposed on the system. This load has been denoted as a standard load profile. Unloading and loading the system can then be made by scaling the loads.

1.6.2 Added loads

In addition the following loads are applied to a grid's standard load profile:

- Local energy communities
- Dedicated storage devices
- Fast charging stations
- A battery powered ferry

Local energy communities can act as one unit, consisting of e.g. an electric power source, storages, electric vehicles (EVs) with vehicle to grid (V2G) capability and households. The ferry further complicates the grid dynamics with its intermittent consumption due to a time-table based arrival and departure.

1.6.3 Charging and discharging

Via charging, voltage dependent loads consume active power from the grid, as they store or produce reactive power. Via discharging, voltage dependent loads supply active power to the grid, as they store or produce reactive power. Meaning, the presence of such loads in the grid, introduces electric noise: directionless power. Their active power contributions are implemented by the user in Algorithm B.2's either feeder- or injection-loop (Section 2.1), as their reactive power contributions are inherently calculated by Algorithm B.3's function *getload*, called by *accload* during a backward sweep (Section 1.3). This thesis investigates the charging of local storages, a ferry and EVs, and the discharging of local storages as backup feeders.

An EV is comparable to a storage battery, but is typically of smaller capacity and mobile. Similarly, the battery powered ferry can be considered a floating battery. The implementation of these added loads is described in Section 4, specifically Sections 4.3, 4.4 and 4.5.

Roughly speaking, a plugged in EV consumes 150 kWh. The grid has a base apparent power of 10 MVA (Section 1.5.2), making the resulting V2G active load equal to 0.015 pu per hour. Distinguishing the battery powered ferry scenario (Section 3.4) from the vehicles to grid scenario (Section 3.5), the ferry's active load was implemented to be double the EV's active load, as seen in Table 1.4. Providing the ferry's on board battery, the onshore battery's discharge is implemented as a negative bus active power injection, depending respectively on its inherent size small, medium or large as seen in Table 1.4. Distinguishing a local storage (Section 3.3) from a ferry's onshore battery, the local storage is implemented to inject one fourth more active power than the onshore battery. Otherwise, for simplicity, they have the same charging slope d^2Q/dV^2. It was fitting to assume that these two battery types were similar, since the market for batteries is quite sparse at the moment.

A battery is not a continuous electric power source, but discharges what the grid consumes with the risk of being depleted, having replaced a feeder currently in an outage. Thus its inherent charging slope needs to be steep to allow for short charging times. PyDSAL simulates a snap-shot of an electrified grid, but the grid's dynamic evolvement is not simulated. Thus a battery's or an EV's capacity is not necessary to identify in this version of the tool. Distinguishing an EV's charging slope from the rest, it's set to equal one quarter of a battery's charging slope, as seen in Table 1.4. Charging for hours on end, the flatter a charging slope, the less reactive power an EV contributes.

Table 1.4: Two electric parameters of voltage dependent loads

Entity	Bus nr.	$(d^2Q/dV^2)^{ref}$ [pu]	P_{inj} [pu]
Local storage	B5, B70, B107 and B115	0.2	-0.04
Battery powered ferry	B124	-	0.03
Onshore storage small	B124	0.2	-0.015
Onshore storage medium	B124	0.2	-0.03
Onshore storage large	B124	0.2	-0.03
EV	B2, B48 and B117	0.05	0.015

The charging slopes are implemented in Algorithm B.2's start-up (Section 2.1), and the change of active power is implemented in its either feeder- or injection-loop (Algorithm B.2's flow chart in Figure 2.2).

PyDSAL has one function each for calculating a battery's and an EV's contribution to voltage control, named *BatteryDroopCrtl* and *V2GDroopCrtl* respectively. These two functions are mentioned in Table 2.1. In effect, they quantify the reactive power contribution of a plugged in voltage dependent load, with the identical equation [Fos20]:

$$\Delta Q^{ctrl} = - \left.\frac{d^2Q}{dV^2}\right|^{ref} V\left(V - V^{ref}\right) \text{[pu]} \tag{2}$$

Equation 2 states that an entity's directionless power contribution equals minus its rate of reactive power change with respect to its voltage $(d^2Q/dV^2)^{ref}$, times its voltage magnitude V, times the difference between its voltage magnitude V and its voltage reference value V^{ref}. For simplicity, the voltage magnitude reference was set to 1.0 pu for all voltage dependent loads.

Considering Equation 2's value, as its first term (the charging slope) is set to be positive in this thesis (Table 1.4):

- If ΔQ^{ctrl} is negative, the last term regarding voltage difference is positive, and the entity transmits directionless power to the grid. Thus the entity has a greater voltage potential than its reference value.
- If $\Delta Q^{ctrl} \to 0$, then $(V - V^{ref}) \to 0$.
- If ΔQ^{ctrl} is positive, the last term regarding voltage difference is negative, and the entity draws directionless power from the grid. Thus the entity has a smaller voltage potential than its reference value.

The entity's minimum reactive power contribution is attained in the second bullet point above.

Downsizing this report, the calculation of the voltage magnitude V is not described. An iteration of a load flow consists of two sweeps (Section 1.3): one backwards followed by one forwards. It is of significance though to point out that the voltage calculation involves two sensitivities if they are not zero, one of type 1 and the other of type 2 (Section 1.4), calculated in the forward sweep to be thus utilized in any reiteration of a load flow. See $BatteryDroopCrtl$ and $V2GDroopCrtl$ in Algorithm B.3 for more details on these sensitivities. Marked with #Ingrid states which of their equations were altered in this thesis, done to attain the correct dimensions (Section 2.2). Now the dimension of the calculated voltage is V, but per-unit normalized. Algorithm B.3's unaltered function $nodeVoltSensSPv2$ is responsible for calculating the sensitivities.

The impact on the voltage changing by injecting ΔQ^{ctrl}, would be different for different nodes as the ratio of resistance and reactance r/x of the distribution system varied. If the resistance is significantly higher than the reactance, it has less impact to change the voltage by injecting ΔQ^{ctrl}, as directionless power yields only to reactance.

2 Software development

PyDSAL is not yet used by electricity companies to analyse their power systems, as the software still does not have a graphical user interface, and otherwise is still under development. The tool's purpose is to solve a load flow. By performing different load flows and comparing their results, the impact of changes in injections and topologies is identified by the user, e.g. as detailed in Section 5.7. However, PyDSAL was missing procedures for systematic extensive calculation on alternative solutions for system operations. The task was then to develop a script to systematically identify the best strategy for solving any upcoming topology update.

To meet the objectives of this thesis, changes to the shell (Section 2.1) and functions (Section 2.2) were required. See Table 1.2 for an overview of the scripts PyDSAL consists of. Minor changes were made to Algorithm B.4, addressed in Section 4.

2.1 Shell development

To perform the different scenario simulations (Section 3), Prof. Fosso's shell (at the end of Algorithm B.3) had to be further developed. Click here to arrive at the previous version of the shell. Thus, a standalone script, from now on referred to as a shell, was written and added to the tool (Table 1.2). In effect, Algorithm B.2 replaces the previous shell version. Avoiding tampering with Algorithm B.3 and remaining focused on developing a new type of shell, were the main reasons for setting Algorithm B.2 apart as an independent entity. The implementation of the simulation scenarios required certain changes to the functions (Table 2.1), tailoring the layout of the simulation results.

Initially, a separate shell was coded for each of the simulation scenarios. Reducing the work load, the individual shells were incorporated into a single systematized one. Thus resulting in a more structured work flow as well as avoiding flow charting multiple shells. Ultimately, the previous shell was elaborated, incorporating these system changes:

- The shell would need to create a new network for almost every case.
- Loads would for some of the cases be added to the network after its creation.
- A feeder change would have to be implemented.
- A battery's discharge and the charging of both a local storage, a ferry and an EV, would require an implementation of change of bus power.
- The simulation outputs were to be properly displayed.

The shell tailored to all scenarios is found in Algorithm B.2. Instead of writing command after command line by line throughout the script, the shell was scripted to systematize the commands, illustrated with a flow chart in Figure 2.2. Thus enabling output to automatically be saved in a systematic manner, as well as providing easier debugging.

Introducing the overall structure of Algorithm B.2, its following five main sequences result in one simulation and its outputs:

1. A scenario, a network and its feeder are chosen.
2. The network is initialized (See flow chart in Figure 2.1).
3. Optional bus power change is applied.
4. An electrified network is simulated.
5. The outputs are stored in lists, displayed either as graphics or tables.

The second sequence establishes the topology. The third sequence enables tweaking of the grid's load profile.

Delving into Algorithm B.2's working parts, its flow chart in Figure 2.2 states four loops:

1. Scenario-loop
2. Network-loop
3. Feeder-loop
4. Injection-loop

The shell begins with linking itself to Algorithm B.3, enabling utilization of the functions. They are to be fed with parameters, thus the parameters for added loads like batteries, ferry and EVs are set at the beginning of the shell. A list of all simulation scenarios to be investigated is provided as input to the shell's first loop, the scenario-loop. In order to configure a network, data from an Excel file is imported in the following network-loop, sorted into two different lists:

- BusList
- LineList

The first containing the system's buses and their parameters, the latter containing its lines and their parameters. At a network-loop's start-up, BusList and LineList is introduced. Thus ensuring that an untouched standard load profile (Section 1.6.1) is processed. Otherwise values from the last case overlap the current one. With every grid configuration change in complying to a case description, changes reflect in both/either BusList and/or LineList (Section 4). Thus as the shell progresses, they are updated, while other lists stay fixed as illustrated in Figure 2.2. A yellow ellipse in Figure 2.2 is an in-/output.

Just before the feeder-loop starts, their latest update is processed. Thus creating an object, the network N, which is a data construct providing a description of all parameters known to a distribution load flow (power flowing in the grid's lines), defining its method of operation. As Algorithm B.2 calls a function of Algorithm B.3 concerning this construct, it is called with the object N. Thus every function called to either mould or extract data from the construct, is scripted with the prefix "N." to its function name. The moulding represent either the network initialization or its load flow simulation, while the extracted data is utilized in displaying the simulation's outputs in tables/graphics.

The grid is ready to be configured after choosing the feeder of the system, thus a start bus is selected. At the feeder-loop's start-up, the chosen feeder determines the start bus number (illustrated with a yellow ellipse in the flow chart in Figure 2.1). See Section 4 for more details on the implementation of every simulation scenario's network. A topology initialization is implemented at the feeder-loop's start-up, as illustrated by the green circle "Initialize topology" in the flow chart in Figure 2.2, with these five steps (functions) illustrated by the flow chart in Figure 2.1:

1. $flatStart$

   resets every bus's electric parameters, visiting every bus in BusList, zeroing its voltage angle, accumulated load and loss, setting its voltage magnitude equal to 1.0 pu.

2. $config3$

   resets every bus's topological parameters with $clearTopology$, followed by visiting every line in LineList, thus updating BusList by tagging each bus with its respective connected lines as well as neighbor buses.

3. $findtree$

   finds a tree structure from the bus number (yellow ellipse in Figure 2.1) it processes, thus updating LineList: From said bus, the line's to and from parameters are switched, setting the positive line flow direction downstream of the feeder. Visiting every line in LineList, every line's flow direction is updated.

4. $config3$

   runs again, executes $clearTopology$, followed by visiting every line in the updated LineList, updating BusList as before.

5. $mainstruct4$

   produces a list of the grid's main branch (Section 1.5.1), starting from the bus number (yellow ellipse in Figure 2.1) it processes, with sublists wherever branching occurs. Thus establishing the topology.

   The list is named TopologyList (yellow ellipse in both Figure 2.1 and Figure 2.2). In effect, $mainstruct4$ trails the single-line diagram (Figure 1.3), enabling FBS (Section 1.3) to cascade this configuration upstream then downstream, and the function $dispTree$ to draw it (Section 2.2.1).

   No changes are made to neither BusList nor LineList.

BusList is required by all the initialization functions, but LineList is required only by two of them: $config3$ and $findtree$.

In summary:

1. $flatStart$

   resets electric parameters of BusList.

2. $config3$

   resets topological parameters of BusList, followed by updating it.

3. $findtree$

   updates line flow directions.

4. $config3$

   repeats procedure, processing updated input.

5. $mainstruct4$

   states the topology with TopologyList.

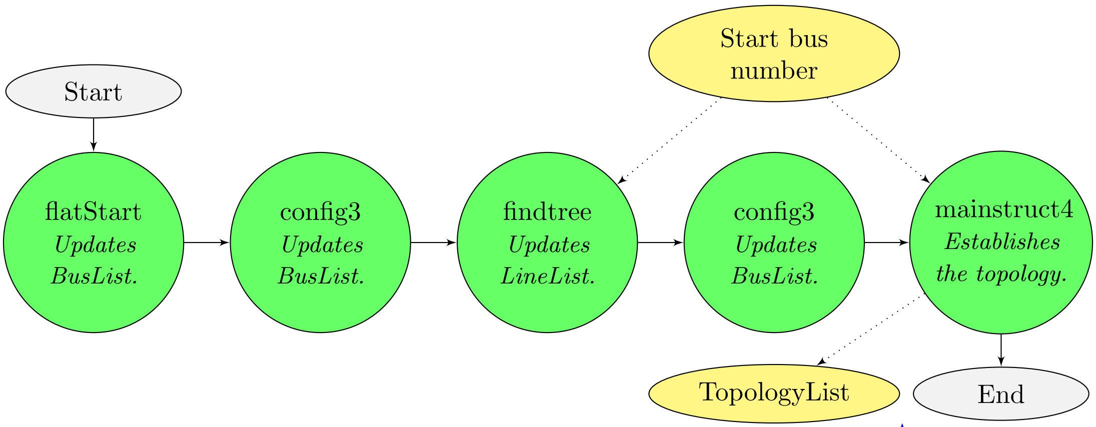


Figure 2.1: Flow chart of a topology's five initialization steps★

★See Figure 2.2 for Algorithm B.2's flow chart, overwriting this flow chart with its green circle "Initialize topology".

With the topology ready to be electrified, any bus injection updates are implemented before executing a simulation. Resetting a grid's load profile following a simulation, BusList is set to its state prior to the optional bus power change, by inverting the bus injection changes. Otherwise, within the injection-loop, a new simulation's load profile is overlapped by the previous load profile.

The systematization of simulations has to do with the shell's backtracking: It is implemented to either rerun or exit a loop. The gray ellipse "Fork" in the flow chart in Figure 2.2 illustrates the shell's forked path following a completed injection-loop, implemented to either resume optional bus power change or skip it. This fork also illustrates the difference between a topology and a simulation: The tool allows multiple simulations to be performed on the same topology only with different load profiles. See Section 6 for future software development on this matter.

This thesis performed a total of nineteen simulations. Thus the injection-loop was executed nineteen times. When all simulation scenarios are completed, no loops are reactivated, and the shell's end-product is three summary tables (Section 5.6, Tables 5.38, 5.39 and 5.40).

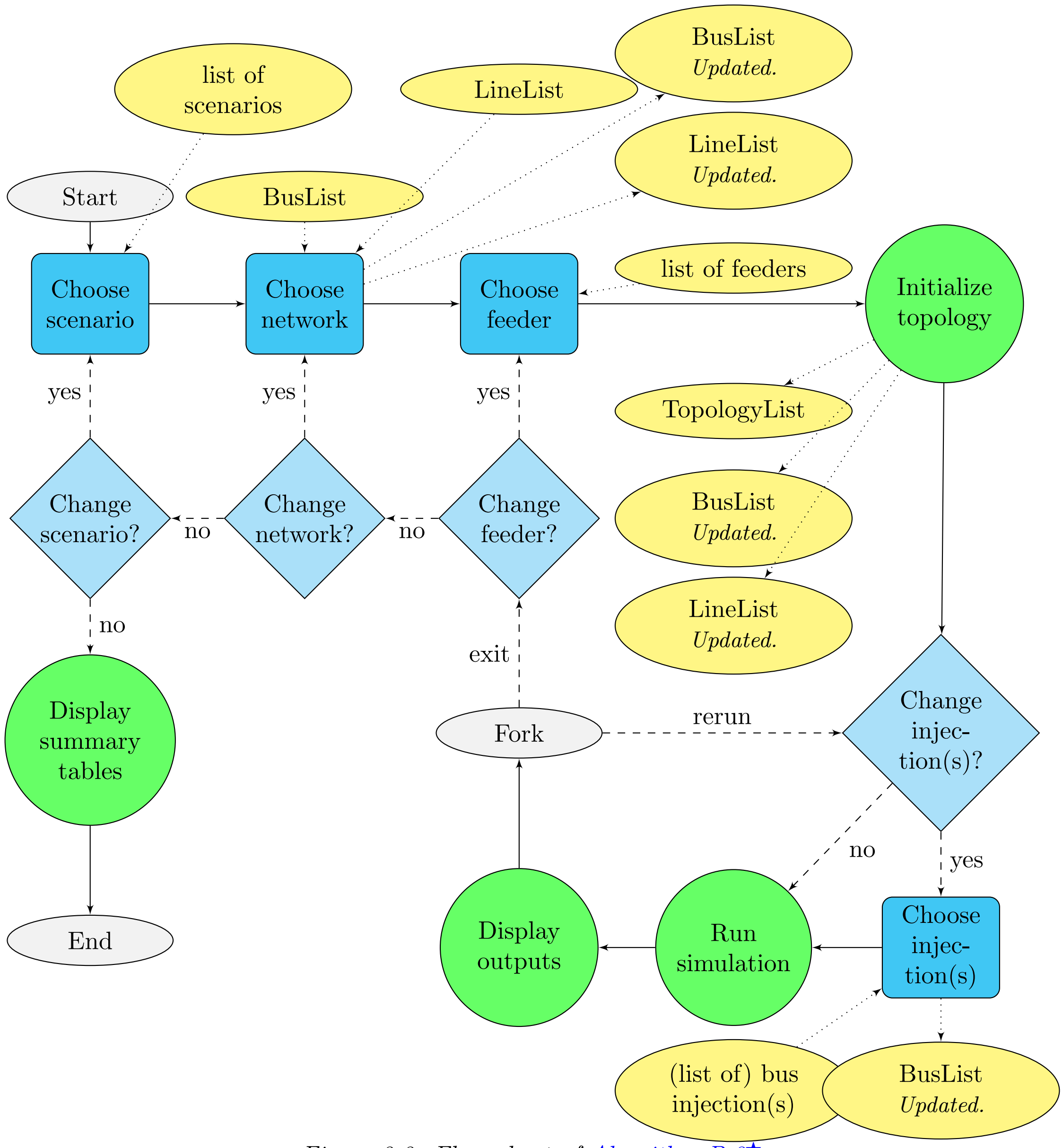


Figure 2.2: Flow chart of Algorithm B.2[★]

[★] See Table 0.2 for this report's terminology.
See Table 1.2 for an overview of PyDSAL's scripts.
See Figure 2.1 for a flow chart of the topology initialization, overwritten by this flow chart's green circle "Initialize topology".
See Figure 2.3 for a flow chart of the drawing of a tree pattern (Section 2.2.1), overwritten by this flow chart's green circle "Display outputs".
See Table 3.1 for an overview of the cases.
See Section 4 for the scenario implementations.
See Table 4.1 for Algorithm B.2's chronological loop record.
See Section 5.7 (Tables 5.38, 5.39 and 5.40) for Algorithm B.2's end product (green circle "Display summary tables").
See Figure 6.2 for a flow chart of a future development of Algorithm B.2.
Click here to arrive at the previous version of PyDSAL's shell.

2.2 Function development

Nine of Algorithm B.3's functions were altered, all adjustments marked #Ingrid. See Table 2.1 for an overview of their tailoring. See Table 2.2 for every tailored function's purpose.

Table 2.1: Algorithm B.3's tailored functions

Function	Main changes	Changes explained
DistLF	Added a command:	Returns a list of a simulation's total power and loss.
BatteryDroopCtrl	Changed an equation:	Changed the second and third coefficients of the differential equation, flipping a ratio. See Section 1.6.3 for more details.
V2GDroopCtrl	Changed an equation:	Changed the second and third coefficients of the differential equation, flipping a ratio. See Section 1.6.3 for more details.
checkFlow	New name: Added a command:	From *checkOverflow* to *checkFlow*. Returns a list of marked flows.
checkVolt	New name: Expanded a command:	From *checkOverLoad* to *checkVolt*. Returns a list of marked voltages.
tableplot	Removed three inputs: Added two inputs: Added color-categories:	*columncol*, *rowcol* and *colw*, regarding column color, row color and column width respectively. A case name and a list of marked flows/voltages. Systematized coloring of the table's first column, ensuring that marked flows/voltages stand out, color-categorized as seen in Table 2.3.
dispFlow	Added two inputs:	A case name and a list of marked flows.
dispVolt	Added two inputs:	A case name and a list of marked voltages.
dispTree	New name: Removed five inputs: Added three inputs:	From *dispGraph* to *dispTree*. The number *top*, and the six lists *feeders*, *LEC*, *charging*, *lowVolt*, *overload*, *disconnected*, regarding distinguishing marked flows and voltages. A case name and two lists *tagBus* and *tagLine*, which are the marked flows and voltages respectively, color-categorized as seen in Table 2.3. See Section 2.2.1 for more details.

Table 2.2: Algorithm B.3's tailored functions' purpose

Function	Purpose
DistLF	Executes a simulation of an electrified distribution grid based on FBS (Section 1.3).
BatteryDroopCtrl	Calculates the battery contribution to voltage control (Section 1.6.3).
V2GDroopCtrl	Calculates the V2G contribution to voltage control (Section 1.6.3).
checkFlow	Categorizes the line flows.
checkVolt	Categorizes the bus voltages.
tableplot	Produces a table.
dispFlow	Calculates and then tabulates a simulation's line flows, then calls *tableplot*.
dispVolt	Tabulates a simulation's bus voltages, then calls *tableplot*.
dispTree	Produces a tree pattern (Section 2.2.1, a color-categorized single-line diagram of an electrified grid, with line flow directions, at a moment in time), stored in an HTML file (Appendix A), enabling zooming.

The layout of simulation outputs was tailored to this report, mainly regarding their color-categorization (Section 2.2.2). This report displays simulation outputs of the electric parameters line flows and bus voltages in tables, while the simulation output illustrating the electrified grid is displayed in a graphic. See Section 2.2.1 for the development on the latter. The tool's previous version was already implemented to produce an HTML file of a grid's color-categorized single-line diagram (Appendix A). Opening such a file in a web browser enables the user to zoom in on areas of interest. The tree graphics used in this report are screenshots of said HTML files.

Previously, every table produced by the tool, created by the function *tableplot*, displayed a maximum of thirteen rows, scripted to have a cyan title column and title row. With a grid containing 124 nodes, this spawned ten tables per parameter outputs. Thus the amount of rows displayed in one table was altered to what is seen in this report, e.g. Table 5.5.

The color-categorizing of a table's title column is introduced, while a graphic's color-categorization was only elaborated. Thus the tables and graphic were synergized. When a bus listed in a table is marked with a distinct color, then it is marked with the same color in the graphic depiction. Thus it's easier to spot the bus and take in its overall part in the scheme. Regretfully, the grid's bus numbering is not visible in the graphic (unless the grid is a subgrid as the single-line diagram in Figure 3.2a), leaving one to rely on the color-categories, seen in Table 2.3.

Also, the calculation of the impact of voltage dependent loads charging in the grid was altered, concerning voltage droop control. This is addressed in Section 1.6.3.

2.2.1 Tree pattern development

Displaying a snap shot of an electrified grid as a color-categorized single-line diagram, materializes a simulation, concretizing PyDSAL's concept. The color-categories are explained in Table 2.3. This snap shot is named "tree pattern". Its objective is to display bus and line numbers, line flow direction arrows and case characteristics. Its line numbering was introduced, implemented in Algorithm B.2. The implementation of its color-categorization was further developed from Algorithm B.1, and is illustrated in the flow chart in Figure 2.3.

The function *dispTree* (mentioned in Tables 2.1 and 2.2) is executed in the shell's injection-loop, within the green circle "Display outputs" in the flow chart in Figure 2.2. In effect, it draws a tree, growing out from a grid's supply node. A tree pattern's feeder has line flow direction arrows leading away from it. With every feeder change the tree growth starts at a new point, drawing a slightly different tree pattern. Meaning, *dispTree* always starts with the node furthest upstream, as it is implemented to process TopologyList, produced by the function *mainstruct* in Algorithm B.2's feeder-loop (Section 2.1).

dispTree processes four inputs, illustrated as yellow ellipses in the flow chart in Figure 2.3:

- The list TopologyList.
- A list of marked lines.
- A list of marked nodes.
- A case name.

TopologyList is produced in the flow chart in Figure 2.2's green cirle "Configure topology", overwriting the flow chart in Figure 2.1.

These inputs are prepared by Algorithm B.2, utilizing some of Algorithm B.3' functions as described in Table 2.1. Thus *dispTree* tags a bus node with its number, duties and voltage category, corresponding a line with its number, flow direction and flow category. These node duties and categories are color-categorized, as explained in Table 2.3.

dispTree begins with creating a graph G. Interpreting this creation as an empty map canvas, the flow chart in Figure 2.3 illustrates the successive pinning of nodes and lines to it. The word "map" is used as a synonym for a tree pattern. In fact, *dispTree* calls two other functions to in turn process G:

1. *AddNodes*

processes TopologyList and the list of marked nodes.

2. *ConnectNodes*

   processes TopologyList and the list of marked lines.

These functions visit every node in TopologyList. In effect, the first adds every node to G, tagging every node, corresponding the latter adds every line to G, tagging every line. Finally, a tree pattern is drawn, establishing G. The case name is put to use when G is saved in an HTML file. The green circle "Display zoomable map" in the flow chart in Figure 2.3 implies that the user opens the HTML file in a web browser. The word "map" is used as a synonym for a tree pattern. A yellow ellipse is an input.

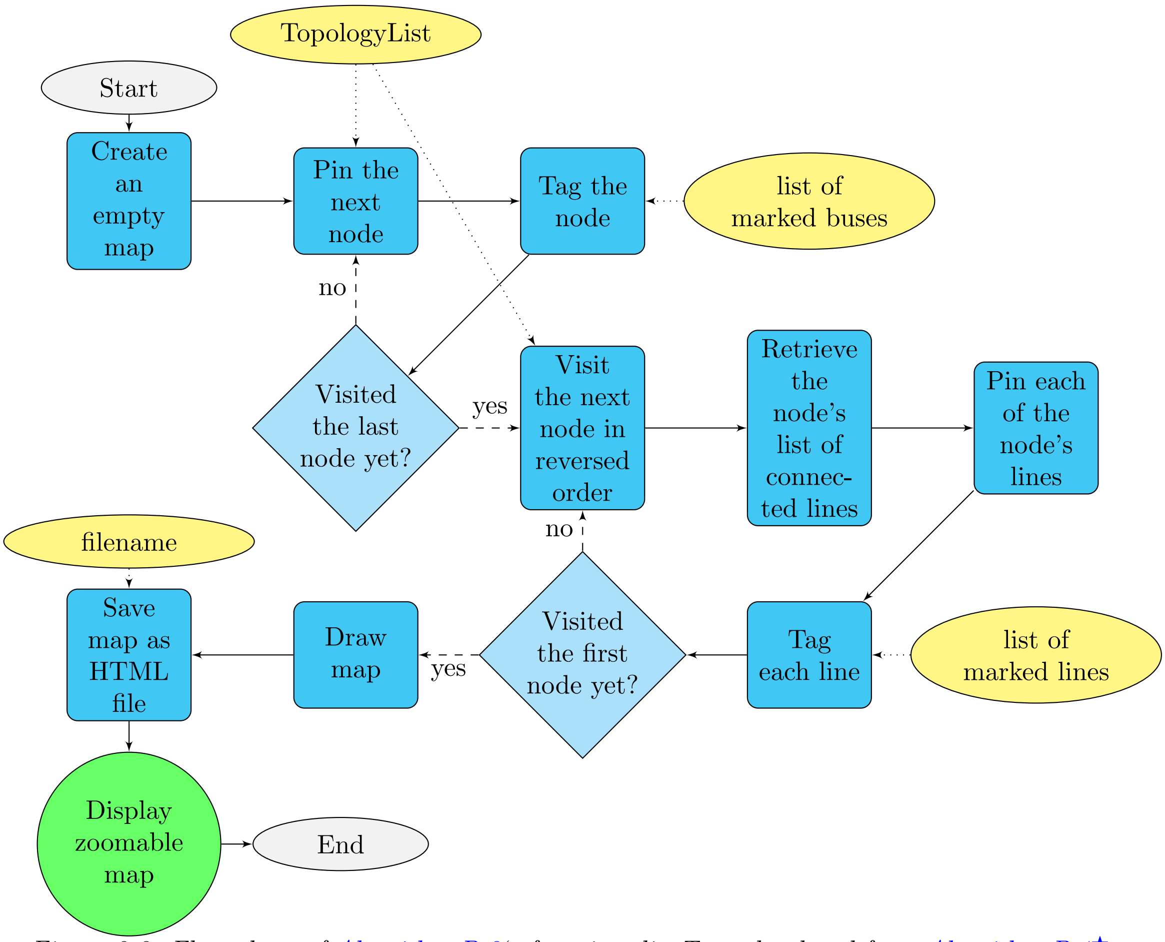


Figure 2.3: Flow chart of Algorithm B.3's function dispTree, developed from Algorithm B.1[★]

A challenge to overcome was implementing a node's color overlap, since it can have only one. Also, implementing a node with multiple tags proved challenging: The first tag would repeat itself. If it is any of the alternative feeders, its green node is larger than the others.

2.2.2 Color-categorization development

The color-categories of simulation results are detailed in Table 2.3. Originally, the idea was to make any preferred imitation of an electrified grid stand out, having simulation outputs with only sea green colored lines and brown nodes. This proved difficult, since this thesis' simulations all are quite similar. Introducing more colors and also widening the categories to chart a simulation's

---

[★] See Figure 2.2 for Algorithm B.2's flow chart, overwriting this flow chart with its green circle "Display outputs".

outputs with, gave a simulation a more distinct ”fingerprint”. Any simulation outputs with red buses/lines illustrate an overloaded grid. More categories than existed in Algorithm B.1 were introduced.

The coloring illustrates a simulation’s characteristics, e.g. Table 2.3 shows that a yellow colored node has a too low voltage magnitude (below or equal to 0.94 pu). This table’s second column has two variables:

- F stands for the percentage of a transmission line flow divided by its line’s transfer capacity.
- V stands for a bus voltage in pu.

Ideally, a node voltage surpasses yellow level, signalling that yellow is an unwanted color. A yellow colored line (transmitting more than 40% and less or equal to 60% of its capacity) doesn’t transmit too little, which might make the color-code counter-intuitive. A yellow line illustrates that it may transmit at least 40% more power than it is currently transmitting before overloading. A line has a higher risk of outage as it operates closer to its max capacity, overheating, giving less room for flow error. Likewise, a too low or too high voltage magnitude would be unsatisfactory for consumers, making their appliances cranky.

Table 2.3: The color-categorization of scenario figures and simulation results explained★

Color	Significant buses	Used in...
green	An alternative feeder	Tree pattern, bus voltages table and scenario figure
cyan	A battery or an EV connected to the bus	Tree pattern, bus voltages table and scenario figure

Color	Line flow [%]	Used in...
red	$F > 100\%$	Tree pattern and line flows table
pink	$80\% < F \leq 100\%$	Tree pattern and line flows table
orange	$60\% < F \leq 80\%$	Tree pattern and line flows table
yellow	$40\% < F \leq 60\%$	Tree pattern and line flows table
seagreen	$0\% < F \leq 40\%$	Tree pattern and line flows table
violet	Zero line flow	Tree pattern and line flows table

Color	Bus voltage [pu]	Used in...
red	$V \geq 1.1$ pu	Tree pattern and bus voltages table
pink	1.0 pu $\leq V < 1.1$ pu	Tree pattern and bus voltages table
brown	0.96 pu $< V < 1.0$ pu	Tree pattern and bus voltages table
orange	0.94 pu $< V \leq 0.96$ pu	Tree pattern and bus voltages table
yellow	$V \leq 0.94$ pu	Tree pattern and bus voltages table
violet	Zero bus voltage	Tree pattern and bus voltages table

PyDSAL is implemented to set the grid’s supply node as the voltage reference for all the other nodes. Thus it is set to have the standard voltage magnitude of 1.0 pu and voltage angle of 0.0. The pink category of voltage magnitudes equal to or greater than 1.0 pu and smaller than 1.1 pu is introduced.

If any node in the grid has a voltage magnitude of exactly 1.0 and a voltage angle exactly of 0.0, this node is interpreted as a supply node by the further developed *tableplot* (Section 2.2), thus bypassing this category, ensuring that the grid’s feeder stands out in the bus voltage table. If the supply node is one of the alternative feeders, it is colored green. If the supply node is one of the local storages, depleting during a feeder’s outage, it is implemented to be colored cyan. The further developed *dispTree* (Section 2.2.1) isn’t implemented to bypass any line flow- or bus voltage-category.

★See Table 5.35 for the indexed simulation results.
See Table 5.36 for the indexed simulation commentaries.
See Tables 5.39 and 5.40 for Algorithm B.2’s color-categorized summary tables.

When an alternative feeders supplies the grid, its green node is in this thesis implemented to be larger than the grid's other nodes, making it stand out. An overloaded supply bus should be red, not green, symbolizing overload. Thus, if any of the buses, even any of the alternative feeders, fell into any of the colored categories, their node color was overlapped by the category's color. Future development on this is discussed in Section 6.

3 Simulation scenarios

This thesis investigates five simulation scenarios, as listed in Table 3.1. Subjecting the grid to these scenarios, it undergoes a total of nineteen cases. Thus nineteen simulations were performed. Their implementation is described in Section 4. Every simulation produces a batch of results, commented on in Section 5. The changes made to the grid in altering it to comply to a case description are discussed in the following subsections.

Every case differs depending on the combination of the grid's feeder, topology and load profile, as detailed by Table 3.1. Cases 1, 5 and 13-19 are all supplied by main feeder B1, but the topology differs. Thus these cases are likely to have near to or even identical results. Changing the feeder causes the simulated power to flow in a different direction. Also, implementing node B1 as the main feeder to supply the grid in the scenarios with a fixed feeder, make the simulations more realistic: Only if the main feeder has an outage are any of the backup feeders supposed to take over the load.

Table 3.1: The grid's nineteen cases[★]

Case	Scenario	Feeder	Topology and any added loads
Case 1	Section 3.1	B1	The original topology
Case 2	Section 3.1	B36	The original topology
Case 3	Section 3.1	B62	The original topology
Case 4	Section 3.1	B88	The original topology
Case 5	Section 3.2	B1	The subgrid left of the disconnected line L16
Case 6	Section 3.2	B36	The subgrid left of the disconnected line L16
Case 7	Section 3.2	B62	The subgrid right of the disconnected line L16
Case 8	Section 3.2	B88	The subgrid right of the disconnected line L16
Case 9	Section 3.3	B5	Four batteries incorporated as provision for feeder outage
Case 10	Section 3.3	B70	Four batteries incorporated as provision for feeder outage
Case 11	Section 3.3	B107	Four batteries incorporated as provision for feeder outage
Case 12	Section 3.3	B115	Four batteries incorporated as provision for feeder outage
Case 13	Section 3.4	B1	One small battery incorporated. The ferry is off grid.
Case 14	Section 3.4	B1	The ferry charges from the incorporated small battery
Case 15	Section 3.4	B1	One medium battery incorporated. The ferry is off grid.
Case 16	Section 3.4	B1	The ferry charges from the incorporated medium battery
Case 17	Section 3.4	B1	One large battery incorporated. The ferry is off grid.
Case 18	Section 3.4	B1	The ferry charges from the incorporated large battery
Case 19	Section 3.5	B1	Three EVs incorporated, all charging.

The first simulation scenario analyses the original topology (Figure 1.3). The others analyse an updated topology, tailored to their respective cases. The objective was to achieve a simulation without any overflowed lines and with bus voltage magnitudes above 0.96 pu and below 1.1 pu.

[★] See Figures 3.1, 3.2a, 3.2b, 3.4, 3.5 and 3.6 for every scenario's single-line diagram.
See Table 5.35 for the indexed simulation results.
See Table 5.37 for a more detailed overview of the cases.
See Figure 2.2 for Algorithm B.2's flow chart.
See Section 4 for the scenario implementations.

3.1 Scenario: Change of supply bus

This scenario is the base scenario, which the other scenarios are referenced to. This scenario's four cases analyse the original topology (Figure 1.3). The objective was to evaluate the alternative feeders impact on supplying the load. The grid has one main feeder and three backup feeders, which are buses B1, B36, B62 and B88 respectively, in green boxes in Figure 3.1.

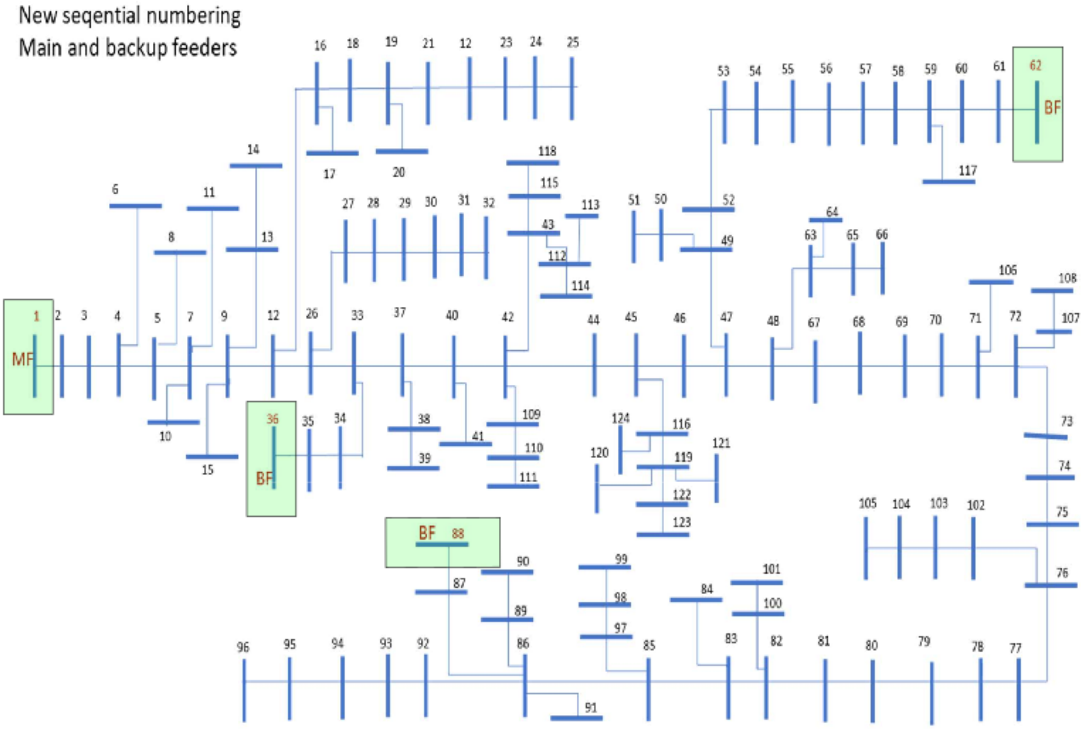


Figure 3.1: Simulation scenario: Change of supply bus★
Alternative feeders in green boxes.

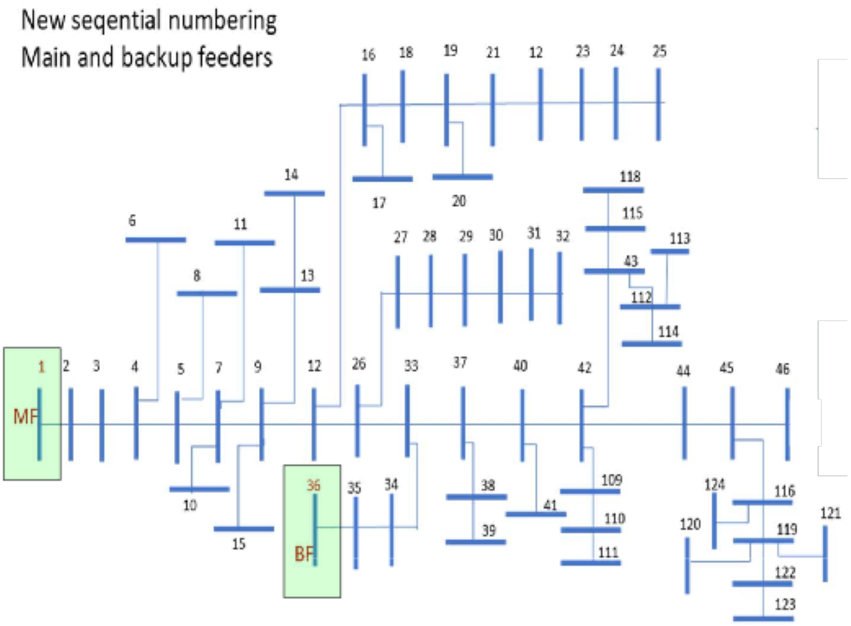
★See Table 3.1 for an overview of the cases.
See Tables 5.2, 5.3, 5.8, 5.9, 5.14, 5.15, 5.20 and 5.21 for this simulation scenario's four sets of line flows, respectively commented on in Tables 5.1, 5.7, 5.13 and 5.19.
See Tables 5.5, 5.11, 5.17 and 5.23 for this simulation scenario's four sets of bus voltages, respectively commented on in Tables 5.4, 5.10, 5.16 and 5.22.
See Figures 5.2, 5.4, 5.6 and 5.8 for this simulation scenario's four tree patterns, respectively commented on in Tables 5.6, 5.12, 5.18 and 5.24.
See Section 4.1 for this scenario implementation.
See Section 5.7 (Tables 5.38, 5.39 and 5.40) for Algorithm B.2's end product (green circle "Display summary tables" in the flow chart in Figure 2.2).

3.2 Scenario: Splitting of the grid

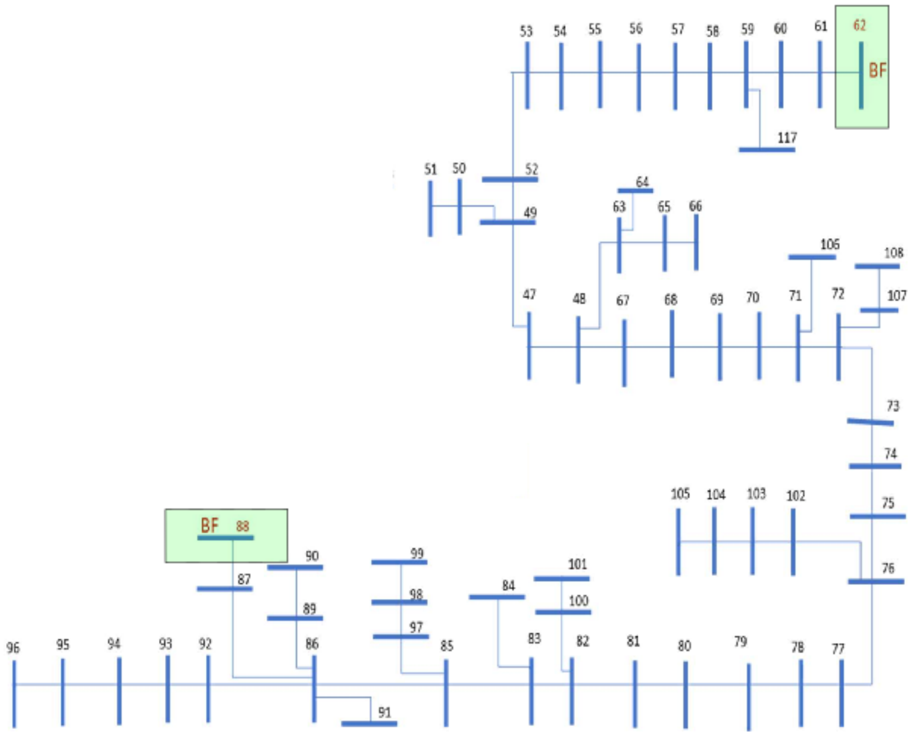
The original topology is split, resulting in two stand-alone subgrids. Two of the cases analyse one subgrid, correspondingly the other two analyse the other subgrid. The cases differ also in that every case has a different feeder. Figure 3.2a shows the subgrid to the left of the split line. Figure 3.2b shows the subgrid to the right of the split line.

With both subnetworks operating independent of each other, each subnetwork's feeder meets a load demand smaller than they are accustomed to. With the feeders less burdened, the lines transmit less power. Since all feeders are connected to the same interconnected grid, the system frequency stays constant despite a feeder change. It is of interest to see whether the grid experiences lower losses operating in split- rather than in standard-mode. The standard-mode, as in Section 3.1, has one feeder supplying the entire grid.

Ideally, subnetworks have equal loading, considering equipment sizes, supply quality and protection gear. The more tailored a grid is, the higher the expense. To split the capacity in half, avoids too steep power swings: If one subnetwork is left with a quarter of the load, while the other subnetwork gets three fourths of the load, then the first experiences steeper power swings than the latter following a splitting of the grid. Operating on half capacity also avoids uneven wear and tear on feeders and other equipment. Another factor to consider is how long this split-mode will be operated for.



(a) Left sub-grid



(b) Right sub-grid

Figure 3.2: Simulation scenario: Splitting of the grid★
Alternative feeders in green boxes.

★See Table 3.1 for an overview of the cases.
See Figures 5.9 and 5.10 for this simulation scenario's left side subgrid's two tree patterns, respectively commented on in Tables 5.25 and 5.26.
See Figures 5.11 and 5.12 for this simulation scenario's right side subgrid's two tree patterns, respectively commented on in Tables 5.27 and 5.28.
See Section 4.2 for this scenario implementation.
See Section 5.7 (Tables 5.38, 5.39 and 5.40) for Algorithm B.2's end product (green circle "Display summary tables" in the flow chart in Figure 2.2).

3.3 Scenario: Local storage as backup feeders

If all the grid's feeders defected, could it still supply itself? With the help of batteries it is possible to keep the grid powered during mainline power cuts. Another benefit of battery solutions is the possibility of charging when both the load and prices are low, and discharge vice versa. Thus the battery could even out peaks in the grid. Using stored power in expensive periods, and charging during cheap periods, would profit the battery's owner. Figure 3.3 shows a typical winter price profile from Nordpool over 24 hours, revealing that night-time charging and discharging at mid-day or dinner-time (18:00) is best.

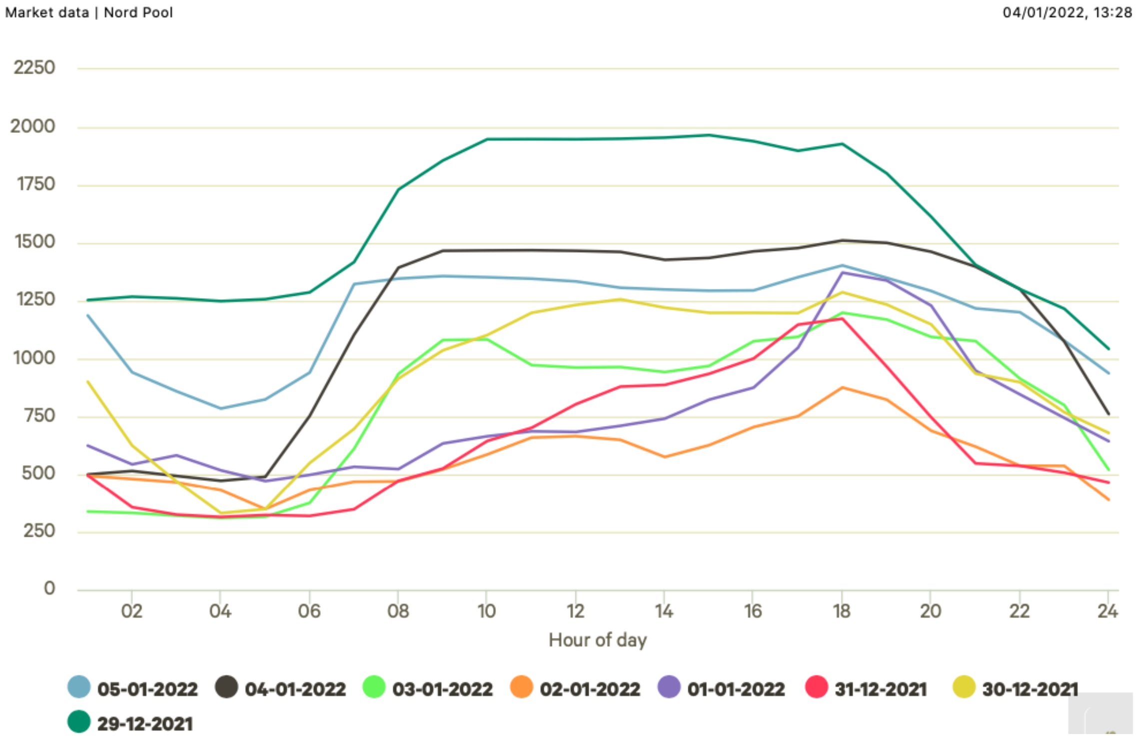


Figure 3.3: Nord Pool hourly price profile [NOK/h] [AS]

Investigating this, batteries were incorporated into the grid. The objective is to evaluate whether a local storage can meet the load demand in a snap-shot of an electrified grid. Four randomly chosen and dispersed buses were assigned an identical battery. Figure 3.4 marks the locations of the randomly dispersed local storages with cyan boxes.

The original topology is updated to incorporate four identical batteries, as provision backup if the feeder was to outage. This scenario's four cases analyse this topology. The cases differ in that every case has a different battery as feeder, as the other three batteries charge. Downsizing this scenario, only four buses were chosen to contain a local storage. Thus this scenario has the same amount of cases as the two preceeding scenarios, which is fitting.

The local storages were dispersed, but consciously chosen not to be located at the end of a branch. Thus further distinguishing this scenario from the base scenario (Section 3.1), as well as making their locations more realistic. A depleting battery in this scenario, feeding the grid, splits its main flow right away or a few lines in, except for bus B5's battery. The buses B70, B107 and B115 are all near the grid's middle, except for bus B5. The latter was of interest, investigating how a provision backup fared compared to main feeder B1. Their flow directions would be quite similar, thus their simulation results should be quite similar.

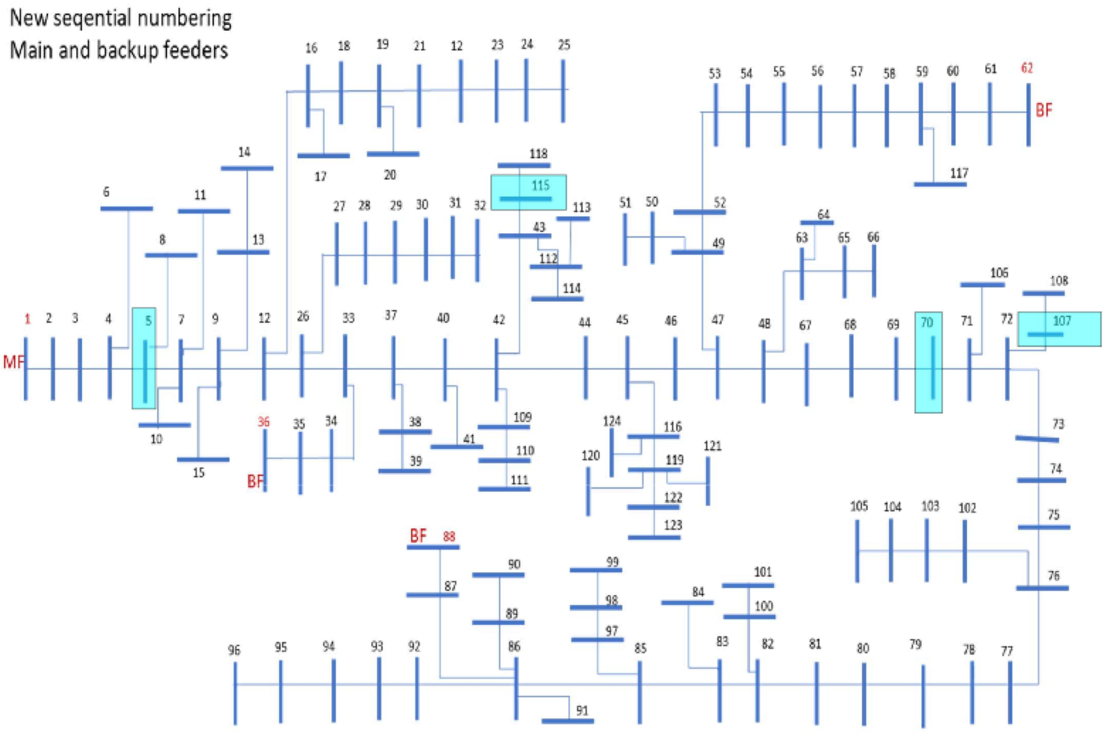


Figure 3.4: Simulation scenario: Local storage as backup feeders[★]
Backup feeders in cyan boxes.

[★] See Table 3.1 for an overview of the cases.
See Figures 5.13, 5.14, 5.15 and 5.16 for this simulation scenario's four tree patterns, respectively commented on in Tables 5.29, 5.30, 5.31 and 5.32.
See Section 4.3 for this scenario implementation.
See Section 5.7 (Tables 5.38, 5.39 and 5.40) for Algorithm B.2's end product (green circle "Display summary tables" in the flow chart in Figure 2.2).

3.4 Scenario: Battery powered ferry

Could the grid sustain an electric ferry consuming power intermittently? Updating the original topology, an onshore battery is incorporated into one randomly chosen bus, feeding the ferry when it docks, charging as the ferry is off grid. Downsizing this scenario, only one bus was chosen to have a dock, and only three different sizes of the onshore battery were chosen. The objective is to investigate the ferry's impact of plugging into the system, and the impact of alternative sizes of the onshore battery. The cyan box in Figure 3.5 marks the location of the onshore battery.

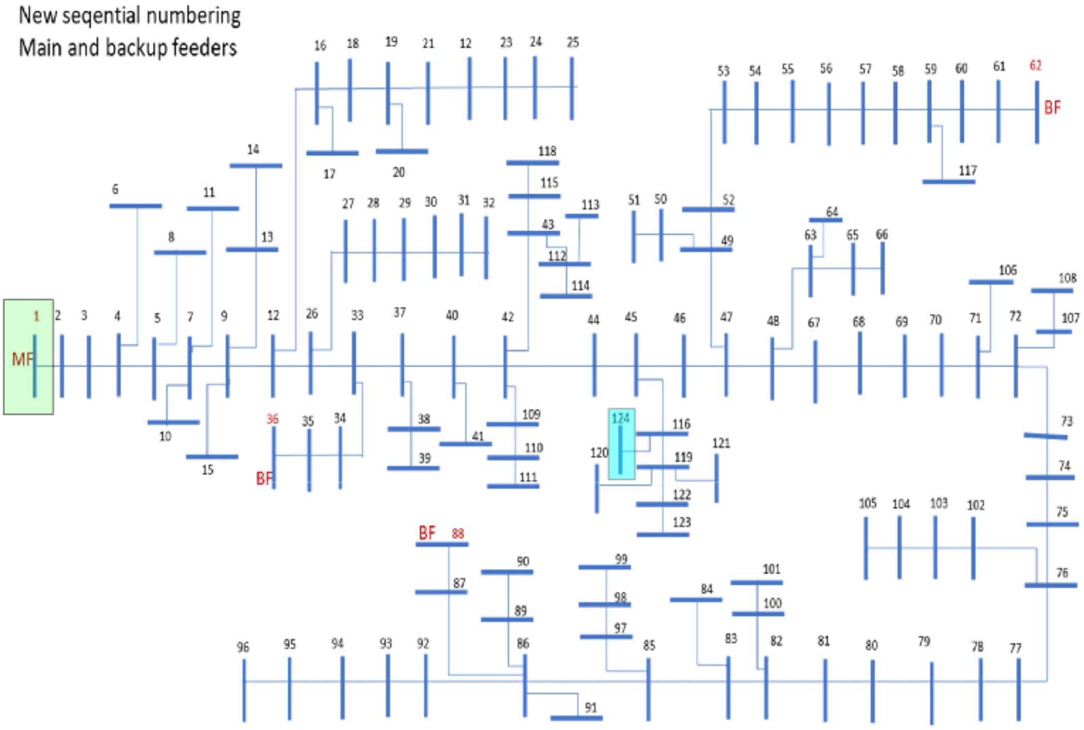


Figure 3.5: Simulation scenario: Battery powered ferry★
Main feeder in green box. Ferry/onshore battery in cyan box.

It was of interest to investigate an onshore battery:

- too small to meet the ferry's demand.
- precisely meeting the ferry's demand.
- meeting the ferry's demand in abundance.

Thus six simulations were performed: Thrice simulating a different onshore battery charging as the ferry is off grid, and correspondingly discharging when the ferry consumes active power. All simulations have main feeder B1 (green box in Figure 3.5) supplying the grid. Thus the simulation results should be quite similar to Case 1's simulation.

★See Table 3.1 for an overview of the cases.
See Figure 5.17 for this simulation scenario's randomly chosen tree pattern, commented on in Table 5.33.
See Section 4.4 for this scenario implementation.
See Section 5.7 (Tables 5.38, 5.39 and 5.40) for Algorithm B.2's end product (green circle "Display summary tables" in the flow chart in Figure 2.2).

When the ferry docks to charge, it can either draw power directly from the grid, with potentially dramatic consequences to the grid's voltage stability, or it can draw the necessary power from an onshore battery. The onshore battery is charged by the grid and only discharges when the ferry docks, which should avoid voltage drops on the grid.

Meeting a ferry's demand, a too small battery will deplete, leaving the grid to overtake the load: Bypassing the onshore battery, the ferry is fed, being the prioritized load. A large battery can charge whilst discharging, since it puts no strain on the system. Thus the battery size is critical, and must be tailored to the ferry's charging time and power consumption in conveying passengers and goods.

3.5 Scenario: Vehicles to grid

Could the grid sustain vehicle to grid (V2G) charging? Updating the original topology, three identical electric vehicles (EVs) were incorporated into the grid. The cyan boxes in Figure 3.6 mark their respective locations. The objective is to discuss the V2G alternative and investigate the impact such a solution has on the grid.

An EV is in this thesis seen as just a power consumption, thus it is fitting to downsize this scenario to just one simulation. Further downsizing this scenario, only three randomly chosen and dispersed buses are assigned an EV: Two on the main branch and one on a subbranch, enabling a viable simulation. Main feeder B1 (green box in Figure 3.6) supplies the system. Thus this scenario's simulation results should be quite similar to Case 1's simulation results.

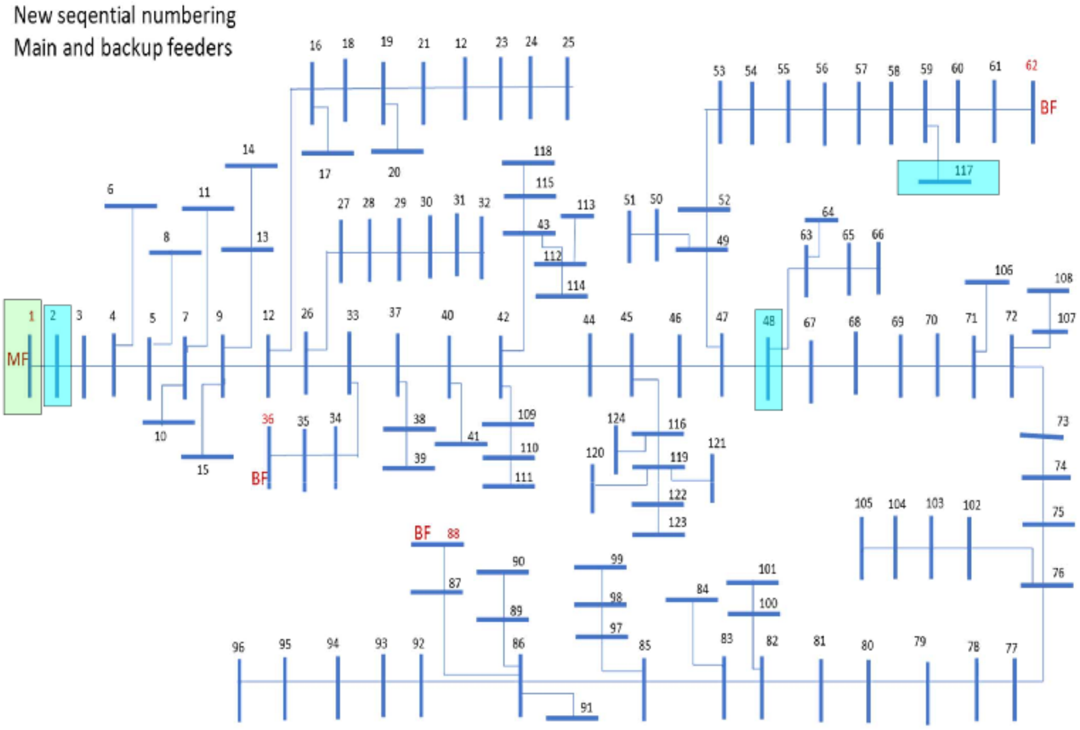


Figure 3.6: Simulation scenario: Vehicles to grid[★]
Main feeder in green box. The electrical vehicles in cyan boxes.

[★] See Table 3.1 for an overview of the cases.
See Figure 5.18 for this simulation scenario's only tree pattern, commented on in Table 5.34.
See Section 4.5 for this scenario implementation.
See Section 5.7 (Tables 5.38, 5.39 and 5.40) for Algorithm B.2's end product (green circle "Display summary tables" in the flow chart in Figure 2.2).

4 Scenario implementation

Algorithm B.2 contains the implementation of the five simulation scenarios. Executing it activates loops in succession, as illustrated by its flow chart in Figure 2.2, as detailed in Table 4.1. This table is discussed in the following subsections. See Section 2.1 for more details.

As seen in Table 3.1, the first two simulation scenarios have no added loads. The rest of the simulation scenarios are implemented to incorporate extra loads to the grid's standard load profile, all implemented to be voltage dependent, except for a ferry's consumption. The latter is implemented as a bus load increase.

The supply is seen as continuous, thus no bus power injection is implemented, except when a local storage discharges during a feeder outage.

The voltage dependent loads (Section 1.6.3) required one new code line to be written for the different load objects. Thus a new parameter *cmode*, control mode, was added to Algorithm B.4's battery and V2G class. A simulation's accumulation of loads is done by the function *accload* (Section 1.2), calling the function *getload* to include any voltage dependent loads present in the grid. Setting *cmode* equal to 2, activated *getload*'s inherent call for voltage droop control, thus calculating the battery's or EV's impact on the system.

Table 4.1: Algorithm B.2's chronological loop record⋆

Case	Scenario-loop	Network-loop	Feeder-loop	Injection-loop
Case 1	run	run	run	run
Case 2	-	-	rerun	run
Case 3	-	-	rerun	run
Case 4	-	-	rerun	run
Case 5	rerun	run	run	run
Case 6	-	-	rerun	run
Case 7	-	-	rerun	run
Case 8	-	-	rerun	run
Case 9	rerun	run	run	run
Case 10	-	-	rerun	run
Case 11	-	-	rerun	run
Case 12	-	-	rerun	run
Case 13	rerun	run	run	run
Case 14	-	-	-	rerun
Case 15	-	rerun	run	run
Case 16	-	-	-	rerun
Case 17	-	rerun	run	run
Case 18	-	-	-	rerun
Case 19	rerun	run	run	run

4.1 ... of a change of supply bus

A network with four alternating feeders was configured (Figure 3.1). Thus four simulations were made. No added loads were incorporated into the topology.

As detailed in Table 4.1, Algorithm B.2 activates all loops in one run, resulting in a simulation of Case 1. Following the exit of the injection-loop, the feeder-loop is reactivated, introducing a new start bus supplying the system, activating the injection-loop, resulting in a simulation of Case 2. This is repeated for Case 3 and Case 4.

⋆See Figure 2.2 for Algorithm B.2's flow chart. See Sections 4.1, 4.2, 4.3, 4.4 and 4.5 for more details on this table.

4.2 ... of a splitting of the grid

In addition to configuring the complete system as one, splitting the system into subparts was configured (Figures 3.2a and 3.2b). The objective was to identify an appropriate separation point to split the grid in different subgrids and supply these from alternative feeders. A separation point was interpreted as a line disconnection, in the same way as a system would be split by a circuit breaker.

The split network was configured by first creating the complete original network (Figure 3.1), then split by updating LineList. This was implemented by splitting the main branch's line L16, which connected buses B46 and B47. In effect, line L16's attribute *ibstat* was zeroed, which is set equal to zero when disconnected, and set to one when connected.

Inspecting the grid in Figure 1.3, splitting it in two between buses B46 and B47, gives the resulting two subnetworks a pair of feeders each. If the split was either between buses B26 and B33 or between buses B47 and B48, one subnetwork would be left more vulnerable than the other, considering one subnetwork gets three feeders while the other only gets one. Presumably the appropriate separation points must be on the main branch stretch between the buses B33 and B47, consisting of seven possible separation points. The grid's "aorta" is split in half, creating two concentrated subgrids rather than one subgrid widespread and the other stumped: The line to the far right of the stretch (L16) was disconnected, downsizing the study to two subgrids with a pair of feeders each and the most equal loading.

This scenario's alternative feeders are the same as in the preceding section, thus four simulations were made. No added loads were incorporated into the topology.

As the flow chart in Figure 2.1 of the five initialization steps of a network illustrates, two of Algorithm B.3's functions update BusList, as detailed in this paragraph. When a line disconnects, the function $config3$ is prohibited from taking it into account. Meaning, the function visits every line in LineList except for the disconnected line. Thus $config3$ is prevented from connecting it to any bus, connecting all other lines to their respective buses, although the other subgrid won't be electrified. The function *mainstruct*4 knits the grid together line by line, visiting the start bus, detecting a neighbor bus, proceeding to knit these two buses together, followed by visiting this neighbor bus, detecting its neighbor bus etc. This start bus is the grid's feeder, furthest upstream in the system, thus introducing the main branch.

Wherever subbranches occur, *mainstruct*4 departs from the main branch, visiting the subbranch, knitting it to completion, followed by departing from it, revisiting the main branch. Visiting the bus previously connected to the line L16, *mainstruct* detects a dead end, followed by departing from this subbranch, revisiting the main branch. Thus the LineList's disconnected line is never knitted into the grid, and *mainstruct*4 never trespasses on the other subgrid. If the grid is initialized with a start bus on the left side, the left side subgrid is configured (Figure 3.2a); and vice versa (Figure 3.2b).

As detailed in Table 4.1, Algorithm B.2 completes the scenario of a change of supply, exiting all loops but the scenario-loop. Reactivating the latter, all loops are activated in one run, resulting in a simulation of Case 5. Following the exit of the injection-loop, the feeder-loop is reactivated, introducing a new start bus supplying the system, activating the injection-loop, resulting in a simulation of Case 6. This is repeated for Case 7 and Case 8.

4.3 ... of local storage as backup feeders

The network containing four identical batteries (Figure 3.4) was configured by updating BusList. Listing them as battery objects, the list BatteryList was scripted. Thus voltage dependent loads were added to the topology. Assigning an object to a bus's attribute *battery*, otherwise set to zero, the buses B5, B70, B107 and B115 were implemented to contain a battery. The network's alternative feeders were implemented to be these local storages. As one battery discharged, implemented as a bus load decrease, supplying the network, as the others charge. Thus four simulations were made. See Section 1.6.3 for the value of the depleting battery's injected power.

As detailed in Table 4.1, Algorithm B.2 completes the scenario of splitting the grid, exiting all loops but the scenario-loop. Reactivating the latter, all loops are activated in one run, resulting in a simulation of Case 9. Following the exit of the injection-loop, the feeder-loop is reactivated, introducing a new start bus supplying the system, activating the injection-loop, resulting in a simulation of Case 10. This is repeated for Case 11 and Case 12.

4.4 ... of a battery powered ferry

The three networks, differing in their onshore battery (Figure 3.5), were configured by updating BusList. All are supplied by main feeder B1. Listing a small, medium and large sized battery object, the list BatterySizes was scripted. Assigning an object to bus B124's attribute *battery*, otherwise set to zero, every network was implemented to contain a different sized battery. Thus a voltage dependent load was added to every topology.

Implementing a stray bus connecting to the grid, in effect adding a bus to the system, received the error "the grid is no longer radial". Discarding this, the ferry's charging was implemented as just a bus load increase: As a change in power consumption rather than a bus with a battery on board re-connecting to the grid. Otherwise, the on board battery would have been taken into account. Providing the ferry's on board battery, the onshore battery's discharge is implemented as a bus load decrease, meeting the ferry's demand. See Section 1.6.3 for the value of both the ferry's and the on shore battery's injected power.

As detailed in Table 4.1, Algorithm B.2 completes the scenario of local storages as backup feeders, exiting all loops but the scenario-loop. Reactivating the latter, all loops are activated in one run, resulting in a simulation of Case 13. Reactivating the injection-loop, Case 14 is simulated. Following the exit of the injection-loop, the feeder-loop is exited as well, followed by reactivating the network-loop, introducing a new topology. The feeder- and injection-loop is activated in one run, resulting in a simulation of Case 15. Reactivating the injection-loop, Case 16 is simulated. This is repeated for Case 17 and Case 18.

4.5 ... of vehicles to grid

The network containing three identical charging EVs (Figure 3.6) was configured by updating BusList. Listing them as V2G objects, the list V2GList was scripted. Thus voltage dependent loads were added to the topology. Assigning an object to a bus's attribute $v2g$, otherwise set to zero, the buses B2, B48 and B117 were implemented to contain an EV. The feeder was set to be main feeder B1. See Section 1.6.3 for the value of the EV's injected power.

As detailed in Table 4.1, Algorithm B.2 completes the scenario of a battery powered ferry, exiting all loops but the scenario-loop. Reactivating the latter, all loops are activated in one run, resulting in a simulation of Case 19.

5 Simulation results

This section comments on Algorithm B.2's simulation outputs (green circle "Display outputs" in the flow chart in Figure 2.2.) Respectively, the simulation outputs commented on in this report display a simulation's:

- tabulated color-categorized line flows.
- tabulated color-categorized bus voltages.
- color-categorized single-line diagram with line flow directions (tree pattern, Section 2.2.1).

5.1 … of a change of supply bus

This simulation scenario has four cases, as seen in Table 3.1. Thus four simulations were performed, as seen in Table 5.35, producing four different sets of line flows, bus voltages and tree patterns.

Case 1 simulation's line flows

This simulation's line flows are seen in Tables 5.2 and 5.3. See Table 5.1 for the commentary on them.

Table 5.1: Commentary on Case 1 simulation's line flows in Tables 5.2 and 5.3[★]

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	L3-L14
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	L65 and L66 (connected to backup feeder B36), L81 and L82, L96 and L97 (connected to backup feeder B62), L120 and L121 (connected to backup feeder B88).

Line flow range	Color	Line(s) or line flow value [pu]
Min. active flow in line(s)	violet	L65, L66, L81, L82, L96, L97, L120 and L121.
Min. active flow in line(s)	seagreen	L95 (This line must have transmitted approximately no flow. If the flow was 0.0 pu, the line would be violet.)
Min. active line flow	violet	0.0 pu
Max. active flow in line	seagreen	L1 (connected to main feeder B1, feeding the grid).
Max. active line flow	seagreen	0.6613 pu

[★] F is a percentage of a line flow divided by its line's capacity. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Table 5.2: Case 1 simulation's line flows for lines L1-L62[★]

	fromBus	toBus	P_{from}	P_{to}	Q_{from}	Q_{to}
L1	1	2	0.6613	-0.6604	0.2248	-0.2236
L2	2	3	0.6502	-0.6497	0.2202	-0.2195
L3	3	4	0.6497	-0.6488	0.2195	-0.2187
L4	4	5	0.6176	-0.6174	0.2085	-0.2083
L5	5	7	0.6103	-0.6088	0.2060	-0.2050
L6	7	9	0.5956	-0.5948	0.2007	-0.2002
L7	9	12	0.5799	-0.5774	0.1953	-0.1937
L8	12	26	0.5501	-0.5484	0.1848	-0.1836
L9	26	33	0.5067	-0.5053	0.1699	-0.1690
L10	33	37	0.4948	-0.4938	0.1656	-0.1650
L11	37	40	0.4879	-0.4859	0.1630	-0.1618
L12	40	42	0.4773	-0.4750	0.1590	-0.1575
L13	42	44	0.4313	-0.4305	0.1431	-0.1426
L14	44	45	0.4274	-0.4272	0.1416	-0.1415
L15	45	46	0.3787	-0.3778	0.1255	-0.1250
L16	46	47	0.3487	-0.3474	0.1154	-0.1146
L17	47	48	0.2467	-0.2466	0.0815	-0.0815
L18	48	67	0.2286	-0.2284	0.0755	-0.0754
L19	67	68	0.2259	-0.2257	0.0746	-0.0744
L20	68	69	0.2248	-0.2244	0.0741	-0.0739
L21	69	70	0.2225	-0.2223	0.0733	-0.0732
L22	70	71	0.2138	-0.2137	0.0704	-0.0703
L23	71	72	0.1934	-0.1933	0.0636	-0.0636
L24	72	73	0.1821	-0.1821	0.0599	-0.0599
L25	73	74	0.1366	-0.1366	0.0449	-0.0449
L26	74	75	0.1366	-0.1365	0.0449	-0.0449
L27	75	76	0.1365	-0.1365	0.0449	-0.0449
L28	76	77	0.0773	-0.0773	0.0254	-0.0254
L29	77	78	0.0773	-0.0773	0.0254	-0.0254
L30	78	79	0.0773	-0.0772	0.0254	-0.0254
L31	79	80	0.0772	-0.0772	0.0254	-0.0254
L32	80	81	0.0751	-0.0751	0.0247	-0.0247
L33	81	82	0.0751	-0.0750	0.0247	-0.0247
L34	82	83	0.0564	-0.0564	0.0186	-0.0186
L35	83	85	0.0537	-0.0537	0.0177	-0.0177
L36	85	86	0.0455	-0.0455	0.0150	-0.0150
L37	86	92	0.0258	-0.0258	0.0085	-0.0085
L38	92	93	0.0184	-0.0184	0.0060	-0.0060
L39	93	94	0.0184	-0.0184	0.0060	-0.0060
L40	94	95	0.0184	-0.0184	0.0060	-0.0060
L41	95	96	0.0184	-0.0184	0.0060	-0.0060
L42	4	6	0.0312	-0.0312	0.0103	-0.0103
L43	5	8	0.0070	-0.0070	0.0023	-0.0023
L44	7	10	0.0077	-0.0077	0.0025	-0.0025
L45	7	11	0.0054	-0.0054	0.0018	-0.0018
L46	9	13	0.0114	-0.0114	0.0038	-0.0038
L47	13	14	0.0045	-0.0045	0.0015	-0.0015
L48	12	16	0.0273	-0.0273	0.0090	-0.0090
L49	16	18	0.0189	-0.0189	0.0062	-0.0062
L50	18	19	0.0130	-0.0130	0.0043	-0.0043
L51	19	21	0.0031	-0.0031	0.0010	-0.0010
L52	21	22	0.0015	-0.0015	0.0005	-0.0005
L53	22	23	0.0015	-0.0015	0.0005	-0.0005
L54	23	24	0.0015	-0.0015	0.0005	-0.0005
L55	24	25	0.0015	-0.0015	0.0005	-0.0005
L56	16	17	0.0084	-0.0084	0.0028	-0.0028
L57	19	20	0.0099	-0.0099	0.0032	-0.0032
L58	26	27	0.0417	-0.0417	0.0137	-0.0137
L59	27	28	0.0417	-0.0417	0.0137	-0.0137
L60	28	29	0.0349	-0.0349	0.0115	-0.0115
L61	29	30	0.0349	-0.0349	0.0115	-0.0115
L62	30	31	0.0349	-0.0349	0.0115	-0.0115

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Table 5.3: Case 1 simulation's line flows for lines L63-L123★

	fromBus	toBus	P_{from}	P_{to}	Q_{from}	Q_{to}
L63	31	32	0.0232	-0.0232	0.0076	-0.0076
L64	33	34	0.0105	-0.0105	0.0034	-0.0034
L65	34	35	0.0000	0.0000	0.0000	0.0000
L66	35	36	0.0000	0.0000	0.0000	0.0000
L67	37	38	0.0059	-0.0059	0.0019	-0.0019
L68	38	39	0.0059	-0.0059	0.0019	-0.0019
L69	40	41	0.0028	-0.0028	0.0009	-0.0009
L70	42	109	0.0323	-0.0323	0.0106	-0.0106
L71	109	110	0.0277	-0.0277	0.0091	-0.0091
L72	110	111	0.0277	-0.0277	0.0091	-0.0091
L73	42	43	0.0114	-0.0114	0.0037	-0.0037
L74	43	115	0.0035	-0.0035	0.0012	-0.0012
L75	115	118	0.0035	-0.0035	0.0012	-0.0012
L76	43	112	0.0079	-0.0079	0.0026	-0.0026
L77	112	113	0.0001	-0.0001	0.0000	-0.0000
L78	112	114	0.0078	-0.0078	0.0026	-0.0026
L79	45	116	0.0485	-0.0485	0.0160	-0.0160
L80	116	119	0.0332	-0.0332	0.0109	-0.0109
L81	119	122	-0.0000	-0.0000	0.0000	0.0000
L82	122	123	0.0000	0.0000	-0.0000	-0.0000
L83	119	121	0.0069	-0.0069	0.0023	-0.0023
L84	116	124	0.0154	-0.0154	0.0050	-0.0050
L85	119	120	0.0263	-0.0263	0.0087	-0.0087
L86	47	49	0.0871	-0.0871	0.0286	-0.0286
L87	49	52	0.0437	-0.0437	0.0144	-0.0144
L88	52	53	0.0437	-0.0437	0.0144	-0.0144
L89	53	54	0.0437	-0.0437	0.0144	-0.0144
L90	54	55	0.0437	-0.0437	0.0144	-0.0144
L91	55	56	0.0121	-0.0121	0.0040	-0.0040
L92	56	57	0.0121	-0.0121	0.0040	-0.0040
L93	57	58	0.0121	-0.0121	0.0040	-0.0040
L94	58	59	0.0121	-0.0121	0.0040	-0.0040
L95	59	60	0.0000	0.0000	-0.0000	-0.0000
L96	60	61	0.0000	0.0000	-0.0000	-0.0000
L97	61	62	0.0000	0.0000	0.0000	0.0000
L98	49	50	0.0075	-0.0075	0.0025	-0.0025
L99	50	51	0.0075	-0.0075	0.0025	-0.0025
L100	59	117	0.0121	-0.0121	0.0040	-0.0040
L101	48	63	0.0180	-0.0180	0.0059	-0.0059
L102	63	65	0.0032	-0.0032	0.0011	-0.0011
L103	65	66	0.0032	-0.0032	0.0011	-0.0011
L104	63	64	0.0148	-0.0148	0.0049	-0.0049
L105	71	106	0.0202	-0.0202	0.0066	-0.0066
L106	72	107	0.0112	-0.0112	0.0037	-0.0037
L107	107	108	0.0112	-0.0112	0.0037	-0.0037
L108	76	102	0.0592	-0.0592	0.0195	-0.0195
L109	102	103	0.0592	-0.0592	0.0195	-0.0195
L110	103	104	0.0296	-0.0296	0.0097	-0.0097
L111	104	105	0.0296	-0.0296	0.0097	-0.0097
L112	82	100	0.0186	-0.0186	0.0061	-0.0061
L113	100	101	0.0186	-0.0186	0.0061	-0.0061
L114	83	84	0.0027	-0.0027	0.0009	-0.0009
L115	85	97	0.0082	-0.0082	0.0027	-0.0027
L116	97	98	0.0082	-0.0082	0.0027	-0.0027
L117	98	99	0.0082	-0.0082	0.0027	-0.0027
L118	86	89	0.0044	-0.0044	0.0014	-0.0014
L119	89	90	0.0044	-0.0044	0.0014	-0.0014
L120	86	87	0.0000	0.0000	0.0000	0.0000
L121	87	88	0.0000	0.0000	0.0000	0.0000
L122	86	91	0.0154	-0.0154	0.0050	-0.0050
L123	9	15	0.0035	-0.0035	0.0011	-0.0011

★Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 1 simulation's bus voltages

This simulation's bus voltages are seen in Table 5.5. See Table 5.4 for the commentary on them.

Table 5.4: Commentary on Case 1 simulation's bus voltages in Table 5.5[★]

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	Main feeder B1 feeds the grid.
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	B47-B108 (coloring the backup feeders B62 and B88 orange) and B117.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category, but the buses B35, backup feeder B36, B60, B61, backup feeder B62, B87, backup feeder B88, B122 and B123 should be violet, since no power is transmitted to them (Table 5.3).
Bus voltage range	Color	Bus(es) or bus voltage magnitude [pu]
Min. voltage at bus	orange	B96, near backup feeder B88. B96 is the bus furthest downstream in the grid.
Min. voltage magnitude	orange	0.95122 pu
Max. voltage at bus	green	Main feeder B1, feeding the grid.
Max. voltage magnitude	green	1.0 pu

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Table 5.5: Case 1 simulation's bus voltages★

	V_{mag}	Θ_V
B1	1.00000	0.00000
B2	0.99836	-0.06937
B3	0.99736	-0.11191
B4	0.99576	-0.15055
B5	0.99536	-0.16015
B6	0.99572	-0.15096
B7	0.99261	-0.19809
B8	0.99536	-0.16016
B9	0.99111	-0.21904
B10	0.99261	-0.19809
B11	0.99261	-0.19809
B12	0.98660	-0.28189
B13	0.99109	-0.21908
B14	0.99107	-0.21912
B15	0.99111	-0.21904
B16	0.98657	-0.28209
B17	0.98657	-0.28209
B18	0.98641	-0.28316
B19	0.98629	-0.28397
B20	0.98629	-0.28397
B21	0.98626	-0.28397
B22	0.98626	-0.28397
B23	0.98626	-0.28387
B24	0.98624	-0.28362
B25	0.98624	-0.28362
B26	0.98313	-0.33070
B27	0.98311	-0.33159
B28	0.98311	-0.33160
B29	0.98311	-0.33160
B30	0.98309	-0.33147
B31	0.98308	-0.33140
B32	0.98305	-0.33252
B33	0.98009	-0.37355
B34	0.98000	-0.37416
B35	0.98000	-0.37416
B36	0.98000	-0.37416
B37	0.97795	-0.40391
B38	0.97793	-0.40404
B39	0.97793	-0.40404
B40	0.97352	-0.46719
B41	0.97350	-0.46707
B42	0.96832	-0.54234
B43	0.96831	-0.54252
B44	0.96631	-0.57148
B45	0.96596	-0.57649
B46	0.96353	-0.61196
B47	0.95962	-0.66933
B48	0.95934	-0.67351
B49	0.95936	-0.67319
B50	0.95931	-0.67289
B51	0.95931	-0.67289
B52	0.95936	-0.67319
B53	0.95902	-0.67317
B54	0.95897	-0.67563
B55	0.95892	-0.67769
B56	0.95892	-0.67769
B57	0.95891	-0.67823
B58	0.95889	-0.67894
B59	0.95889	-0.67894
B60	0.95889	-0.67894
B61	0.95889	-0.67894
B62	0.95889	-0.67894
B63	0.95889	-0.67272
B64	0.95889	-0.67273
B65	0.95885	-0.67302
B66	0.95885	-0.67302
B67	0.95838	-0.68763
B68	0.95712	-0.70614
B69	0.95549	-0.73030
B70	0.95466	-0.74270
B71	0.95413	-0.75055
B72	0.95334	-0.76225
B73	0.95333	-0.76240
B74	0.95332	-0.76256
B75	0.95275	-0.76397
B76	0.95275	-0.76398
B77	0.95275	-0.76398
B78	0.95275	-0.76398
B79	0.95221	-0.76533
B80	0.95220	-0.76537
B81	0.95220	-0.76542
B82	0.95213	-0.76647
B83	0.95180	-0.77137
B84	0.95180	-0.77137
B85	0.95155	-0.77503
B86	0.95135	-0.77807
B87	0.95135	-0.77807
B88	0.95135	-0.77807
B89	0.95135	-0.77813
B90	0.95134	-0.77817
B91	0.95134	-0.77860
B92	0.95135	-0.77807
B93	0.95135	-0.77807
B94	0.95130	-0.77953
B95	0.95129	-0.77955
B96	0.95122	-0.77909
B97	0.95147	-0.77477
B98	0.95147	-0.77485
B99	0.95147	-0.77485
B100	0.95205	-0.76597
B101	0.95205	-0.76597
B102	0.95275	-0.76398
B103	0.95271	-0.76408
B104	0.95267	-0.76418
B105	0.95267	-0.76418
B106	0.95410	-0.75039
B107	0.95330	-0.76295
B108	0.95329	-0.76296
B109	0.96818	-0.54326
B110	0.96805	-0.54413
B111	0.96805	-0.54415
B112	0.96828	-0.54327
B113	0.96828	-0.54327
B114	0.96816	-0.54408
B115	0.96829	-0.54285
B116	0.96585	-0.57727
B117	0.95889	-0.67894
B118	0.96829	-0.54285
B119	0.96578	-0.57776
B120	0.96566	-0.57704
B121	0.96576	-0.57810
B122	0.96578	-0.57776
B123	0.96578	-0.57776
B124	0.96585	-0.57727

★Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 1 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.2. See Table 5.6 for the commentary on it.

Table 5.6: Commentary on Case 1 simulation's tree pattern in Figure 5.2[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the grid, which is main feeder B1. The grid has four alternative feeders, only two of them are colored green. The other two were colored orange. The smaller green node is backup feeder B36.
A battery or an EV	cyan	None in this category.
Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	A string of twelve lines on the main branch, nearly furthest upstream, only two lines apart from the feeder.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A string of two lines doesn't transmit to backup feeder B36, a string of two lines doesn't transmit to bus B123, a string of two lines doesn't transmit to backup feeder B62, and a string of two lines doesn't transmit to backup feeder B88.
Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
$1.0 \text{ pu} \leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the grid's feeder, main feeder B1, by default set with a voltage of 1.0 pu
$0.96 \text{ pu} < V < 1.0$ pu	brown	The rest of the buses.
$0.94 \text{ pu} < V \leq 0.96$ pu	orange	Approximately half of the grid's 124 nodes are colored orange, all connected furthest downstream of the grid (coloring the backup feeders B62 and B88 orange, both connected to violet lines).
$0.0 \text{ pu} < V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

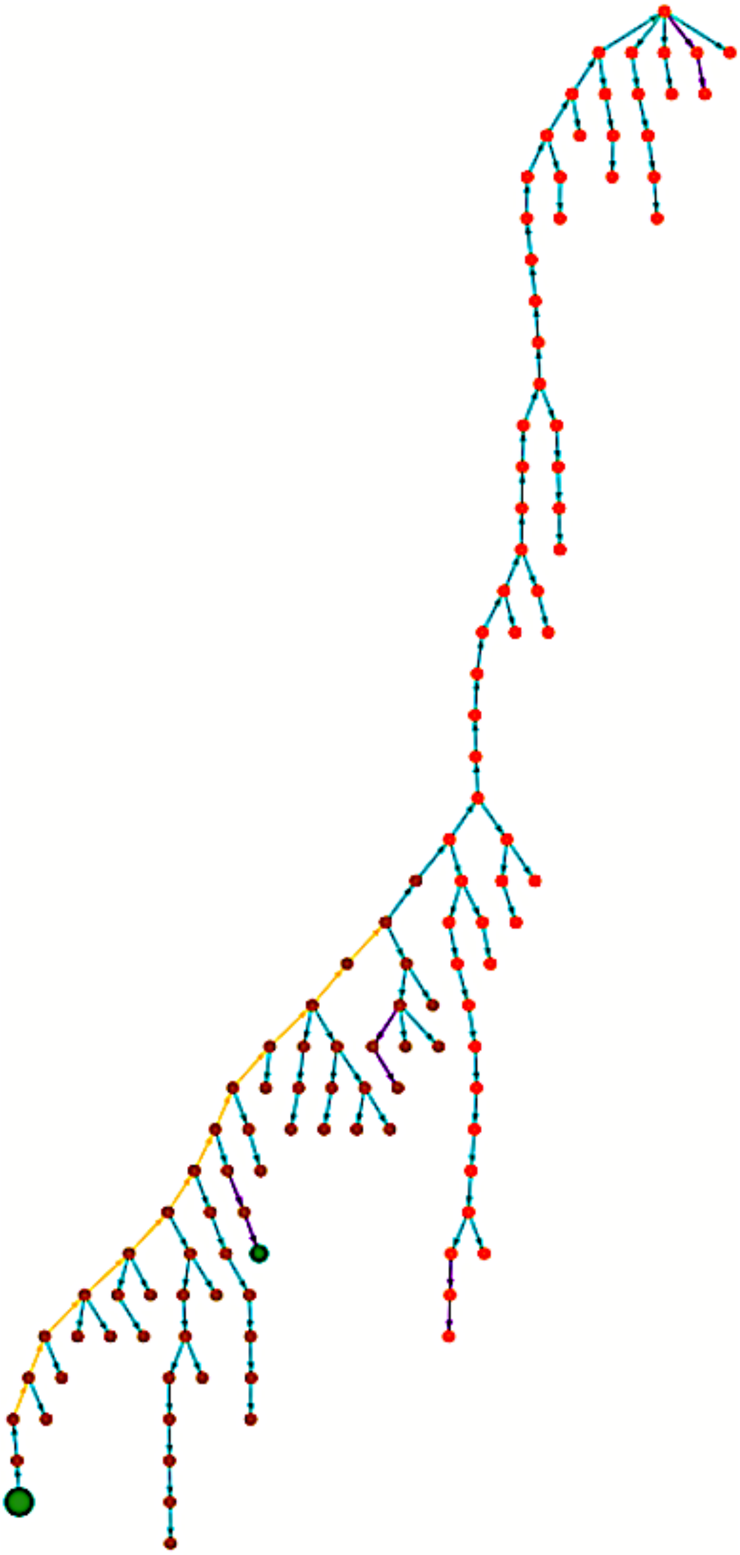


Figure 5.2: Case 1 simulation's tree pattern[★]
Main feeder B1 as feeder.

[★] See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 2 simulation's line flows

This simulation's line flows are seen in Tables 5.8 and 5.9. See Table 5.7 for the commentary on them.

Table 5.7: Commentary on Case 2 simulation's line flows in Tables 5.8 and 5.9★

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	L64 and L65.
$40\% < F \leq 60\%$	yellow	L10-L14
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	L1 (connected to main feeder B1), L81 and L82, L95 and L96 (L97 is connected to backup feeder B62), L120 and L121 (connected to backup feeder B88).

Line flow range	Color	Line(s) or line flow value [pu]
Min. active flow in line(s)	violet	L1, L81, L82, L95, L96, L120 and L121.
Min. active flow in line(s)	seagreen	L97 (connected to backup feeder B62), probably not exactly zero, thus not tagged violet.
Min. active line flow	violet	0.0 pu
Max. active flow in line	seagreen	L66 (connected to backup feeder B36, feeding the grid).
Max. active line flow	seagreen	0.6561 pu

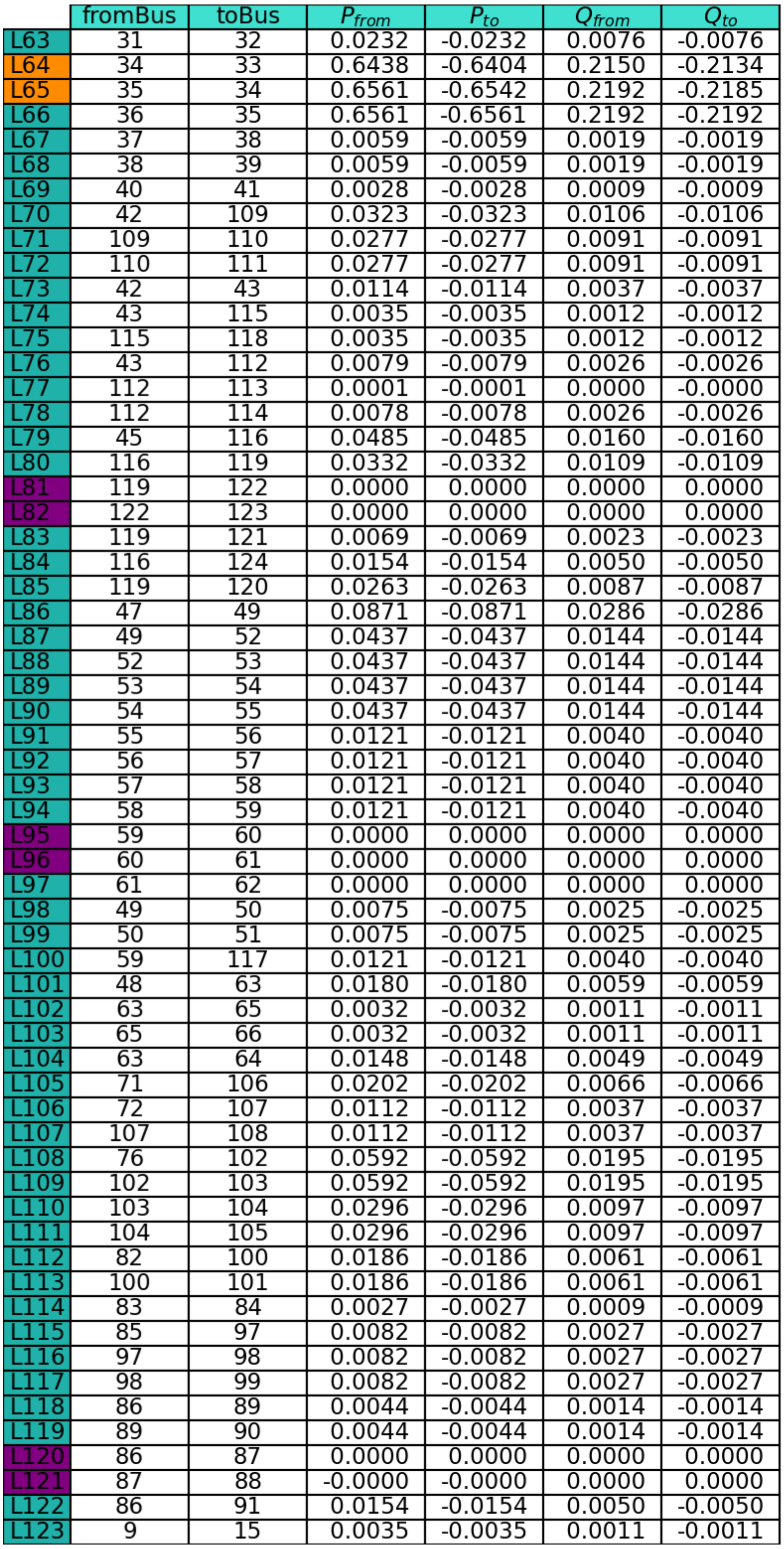
★F is a percentage of a line flow divided by its line's capacity. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Table 5.8: Case 2 simulation's line flows for lines L1-L62[★]

	fromBus	toBus	P_{from}	P_{to}	Q_{from}	Q_{to}
L1	2	1	0.0000	0.0000	0.0000	0.0000
L2	3	2	0.0103	-0.0103	0.0034	-0.0034
L3	4	3	0.0103	-0.0103	0.0034	-0.0034
L4	5	4	0.0415	-0.0415	0.0136	-0.0136
L5	7	5	0.0485	-0.0485	0.0159	-0.0159
L6	9	7	0.0617	-0.0617	0.0203	-0.0203
L7	12	9	0.0767	-0.0766	0.0252	-0.0252
L8	26	12	0.1040	-0.1040	0.0342	-0.0342
L9	33	26	0.1458	-0.1457	0.0480	-0.0479
L10	33	37	0.4946	-0.4936	0.1654	-0.1648
L11	37	40	0.4877	-0.4857	0.1629	-0.1617
L12	40	42	0.4772	-0.4749	0.1589	-0.1574
L13	42	44	0.4312	-0.4304	0.1431	-0.1426
L14	44	45	0.4273	-0.4272	0.1415	-0.1414
L15	45	46	0.3786	-0.3778	0.1255	-0.1250
L16	46	47	0.3486	-0.3474	0.1154	-0.1146
L17	47	48	0.2467	-0.2466	0.0815	-0.0814
L18	48	67	0.2286	-0.2284	0.0755	-0.0754
L19	67	68	0.2259	-0.2256	0.0746	-0.0744
L20	68	69	0.2248	-0.2244	0.0741	-0.0739
L21	69	70	0.2225	-0.2223	0.0733	-0.0732
L22	70	71	0.2138	-0.2136	0.0703	-0.0703
L23	71	72	0.1934	-0.1933	0.0636	-0.0636
L24	72	73	0.1821	-0.1821	0.0599	-0.0599
L25	73	74	0.1366	-0.1366	0.0449	-0.0449
L26	74	75	0.1366	-0.1365	0.0449	-0.0449
L27	75	76	0.1365	-0.1365	0.0449	-0.0449
L28	76	77	0.0773	-0.0773	0.0254	-0.0254
L29	77	78	0.0773	-0.0773	0.0254	-0.0254
L30	78	79	0.0773	-0.0772	0.0254	-0.0254
L31	79	80	0.0772	-0.0772	0.0254	-0.0254
L32	80	81	0.0751	-0.0751	0.0247	-0.0247
L33	81	82	0.0751	-0.0750	0.0247	-0.0247
L34	82	83	0.0564	-0.0564	0.0186	-0.0186
L35	83	85	0.0537	-0.0537	0.0177	-0.0177
L36	85	86	0.0455	-0.0455	0.0150	-0.0150
L37	86	92	0.0258	-0.0258	0.0085	-0.0085
L38	92	93	0.0184	-0.0184	0.0060	-0.0060
L39	93	94	0.0184	-0.0184	0.0060	-0.0060
L40	94	95	0.0184	-0.0184	0.0060	-0.0060
L41	95	96	0.0184	-0.0184	0.0060	-0.0060
L42	4	6	0.0312	-0.0312	0.0103	-0.0103
L43	5	8	0.0070	-0.0070	0.0023	-0.0023
L44	7	10	0.0077	-0.0077	0.0025	-0.0025
L45	7	11	0.0054	-0.0054	0.0018	-0.0018
L46	9	13	0.0114	-0.0114	0.0038	-0.0038
L47	13	14	0.0045	-0.0045	0.0015	-0.0015
L48	12	16	0.0273	-0.0273	0.0090	-0.0090
L49	16	18	0.0189	-0.0189	0.0062	-0.0062
L50	18	19	0.0130	-0.0130	0.0043	-0.0043
L51	19	21	0.0031	-0.0031	0.0010	-0.0010
L52	21	22	0.0015	-0.0015	0.0005	-0.0005
L53	22	23	0.0015	-0.0015	0.0005	-0.0005
L54	23	24	0.0015	-0.0015	0.0005	-0.0005
L55	24	25	0.0015	-0.0015	0.0005	-0.0005
L56	16	17	0.0084	-0.0084	0.0028	-0.0028
L57	19	20	0.0099	-0.0099	0.0032	-0.0032
L58	26	27	0.0417	-0.0417	0.0137	-0.0137
L59	27	28	0.0417	-0.0417	0.0137	-0.0137
L60	28	29	0.0349	-0.0349	0.0115	-0.0115
L61	29	30	0.0349	-0.0349	0.0115	-0.0115
L62	30	31	0.0349	-0.0349	0.0115	-0.0115

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Table 5.9: Case 2 simulation's line flows for lines L63-L123[★]



	fromBus	toBus	P_{from}	P_{to}	Q_{from}	Q_{to}
L63	31	32	0.0232	-0.0232	0.0076	-0.0076
L64	34	33	0.6438	-0.6404	0.2150	-0.2134
L65	35	34	0.6561	-0.6542	0.2192	-0.2185
L66	36	35	0.6561	-0.6561	0.2192	-0.2192
L67	37	38	0.0059	-0.0059	0.0019	-0.0019
L68	38	39	0.0059	-0.0059	0.0019	-0.0019
L69	40	41	0.0028	-0.0028	0.0009	-0.0009
L70	42	109	0.0323	-0.0323	0.0106	-0.0106
L71	109	110	0.0277	-0.0277	0.0091	-0.0091
L72	110	111	0.0277	-0.0277	0.0091	-0.0091
L73	42	43	0.0114	-0.0114	0.0037	-0.0037
L74	43	115	0.0035	-0.0035	0.0012	-0.0012
L75	115	118	0.0035	-0.0035	0.0012	-0.0012
L76	43	112	0.0079	-0.0079	0.0026	-0.0026
L77	112	113	0.0001	-0.0001	0.0000	-0.0000
L78	112	114	0.0078	-0.0078	0.0026	-0.0026
L79	45	116	0.0485	-0.0485	0.0160	-0.0160
L80	116	119	0.0332	-0.0332	0.0109	-0.0109
L81	119	122	0.0000	0.0000	0.0000	0.0000
L82	122	123	0.0000	0.0000	0.0000	0.0000
L83	119	121	0.0069	-0.0069	0.0023	-0.0023
L84	116	124	0.0154	-0.0154	0.0050	-0.0050
L85	119	120	0.0263	-0.0263	0.0087	-0.0087
L86	47	49	0.0871	-0.0871	0.0286	-0.0286
L87	49	52	0.0437	-0.0437	0.0144	-0.0144
L88	52	53	0.0437	-0.0437	0.0144	-0.0144
L89	53	54	0.0437	-0.0437	0.0144	-0.0144
L90	54	55	0.0437	-0.0437	0.0144	-0.0144
L91	55	56	0.0121	-0.0121	0.0040	-0.0040
L92	56	57	0.0121	-0.0121	0.0040	-0.0040
L93	57	58	0.0121	-0.0121	0.0040	-0.0040
L94	58	59	0.0121	-0.0121	0.0040	-0.0040
L95	59	60	0.0000	0.0000	0.0000	0.0000
L96	60	61	0.0000	0.0000	0.0000	0.0000
L97	61	62	0.0000	0.0000	0.0000	0.0000
L98	49	50	0.0075	-0.0075	0.0025	-0.0025
L99	50	51	0.0075	-0.0075	0.0025	-0.0025
L100	59	117	0.0121	-0.0121	0.0040	-0.0040
L101	48	63	0.0180	-0.0180	0.0059	-0.0059
L102	63	65	0.0032	-0.0032	0.0011	-0.0011
L103	65	66	0.0032	-0.0032	0.0011	-0.0011
L104	63	64	0.0148	-0.0148	0.0049	-0.0049
L105	71	106	0.0202	-0.0202	0.0066	-0.0066
L106	72	107	0.0112	-0.0112	0.0037	-0.0037
L107	107	108	0.0112	-0.0112	0.0037	-0.0037
L108	76	102	0.0592	-0.0592	0.0195	-0.0195
L109	102	103	0.0592	-0.0592	0.0195	-0.0195
L110	103	104	0.0296	-0.0296	0.0097	-0.0097
L111	104	105	0.0296	-0.0296	0.0097	-0.0097
L112	82	100	0.0186	-0.0186	0.0061	-0.0061
L113	100	101	0.0186	-0.0186	0.0061	-0.0061
L114	83	84	0.0027	-0.0027	0.0009	-0.0009
L115	85	97	0.0082	-0.0082	0.0027	-0.0027
L116	97	98	0.0082	-0.0082	0.0027	-0.0027
L117	98	99	0.0082	-0.0082	0.0027	-0.0027
L118	86	89	0.0044	-0.0044	0.0014	-0.0014
L119	89	90	0.0044	-0.0044	0.0014	-0.0014
L120	86	87	0.0000	0.0000	0.0000	0.0000
L121	87	88	-0.0000	-0.0000	0.0000	0.0000
L122	86	91	0.0154	-0.0154	0.0050	-0.0050
L123	9	15	0.0035	-0.0035	0.0011	-0.0011

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 2 simulation's bus voltages

This simulation's bus voltages are seen in Table 5.11. See Table 5.10 for the commentary on them.

Table 5.10: Commentary on Case 2 simulation's bus voltages in Table 5.11[★]

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category, but bus B35 (one line downstream of backup feeder B36, feeding the grid) should be colored pink as it has a voltage magnitude of 1.0 pu.
$V = 1.0$ pu	green	Backup feeder B36 feeds the grid.
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	None in this category.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category, but the buses main feeder B1, B60, B61, backup feeder B62, B87, backup feeder B88, B122 and B123 should be violet, since no power is transmitted to them (Tables 5.8 and 5.9).

Bus voltage range	Color	Bus(es) or bus voltage magnitude [pu]
Min. voltage at bus	brown	B96, near backup feeder B88. B96 is the bus furthest downstream in the grid.
Min. voltage magnitude	brown	0.96308 pu
Max. voltage at bus	green	Backup feeder B36, feeding the grid.
Max. voltage magnitude	green	1.0 pu

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Table 5.11: Case 2 simulation's bus voltages★

	V_{mag}	Θ_V
B1	0.98905	-0.07864
B2	0.98905	-0.07864
B3	0.98907	-0.07795
B4	0.98910	-0.07732
B5	0.98912	-0.07666
B6	0.98906	-0.07774
B7	0.98934	-0.07352
B8	0.98912	-0.07666
B9	0.98950	-0.07128
B10	0.98934	-0.07352
B11	0.98934	-0.07352
B12	0.99009	-0.06276
B13	0.98948	-0.07132
B14	0.98946	-0.07136
B15	0.98950	-0.07128
B16	0.99006	-0.06296
B17	0.99006	-0.06296
B18	0.98990	-0.06403
B19	0.98978	-0.06483
B20	0.98978	-0.06483
B21	0.98975	-0.06483
B22	0.98975	-0.06483
B23	0.98975	-0.06473
B24	0.98973	-0.06448
B25	0.98973	-0.06448
B26	0.99074	-0.05343
B27	0.99072	-0.05431
B28	0.99072	-0.05431
B29	0.99072	-0.05431
B30	0.99070	-0.05418
B31	0.99069	-0.05411
B32	0.99067	-0.05522
B33	0.99161	-0.04108
B34	0.99712	-0.00605
B35	1.00000	-0.00002
B36	1.00000	0.00000
B37	0.98949	-0.07073
B38	0.98947	-0.07086
B39	0.98947	-0.07086
B40	0.98512	-0.13254
B41	0.98510	-0.13242
B42	0.97997	-0.20592
B43	0.97996	-0.20610
B44	0.97799	-0.23437
B45	0.97765	-0.23926
B46	0.97525	-0.27389
B47	0.97138	-0.32988
B48	0.97110	-0.33396
B49	0.97112	-0.33365
B50	0.97107	-0.33335
B51	0.97107	-0.33335
B52	0.97112	-0.33365
B53	0.97079	-0.33363
B54	0.97074	-0.33603
B55	0.97069	-0.33804
B56	0.97069	-0.33804
B57	0.97068	-0.33857
B58	0.97066	-0.33926
B59	0.97066	-0.33926
B60	0.97066	-0.33926
B61	0.97066	-0.33926
B62	0.97066	-0.33926
B63	0.97066	-0.33319
B64	0.97066	-0.33320
B65	0.97062	-0.33348
B66	0.97062	-0.33348
B67	0.97016	-0.34774
B68	0.96892	-0.36580
B69	0.96730	-0.38938
B70	0.96648	-0.40147
B71	0.96596	-0.40913
B72	0.96518	-0.42055
B73	0.96517	-0.42069
B74	0.96516	-0.42085
B75	0.96460	-0.42223
B76	0.96460	-0.42224
B77	0.96460	-0.42224
B78	0.96460	-0.42224
B79	0.96406	-0.42355
B80	0.96406	-0.42359
B81	0.96405	-0.42364
B82	0.96398	-0.42467
B83	0.96366	-0.42944
B84	0.96366	-0.42944
B85	0.96342	-0.43301
B86	0.96321	-0.43598
B87	0.96321	-0.43598
B88	0.96321	-0.43598
B89	0.96321	-0.43605
B90	0.96321	-0.43608
B91	0.96320	-0.43650
B92	0.96321	-0.43598
B93	0.96321	-0.43598
B94	0.96316	-0.43741
B95	0.96315	-0.43743
B96	0.96308	-0.43698
B97	0.96333	-0.43276
B98	0.96333	-0.43284
B99	0.96333	-0.43284
B100	0.96390	-0.42417
B101	0.96390	-0.42417
B102	0.96460	-0.42224
B103	0.96455	-0.42234
B104	0.96452	-0.42243
B105	0.96452	-0.42243
B106	0.96593	-0.40898
B107	0.96513	-0.42123
B108	0.96513	-0.42124
B109	0.97984	-0.20682
B110	0.97971	-0.20767
B111	0.97971	-0.20768
B112	0.97993	-0.20683
B113	0.97993	-0.20683
B114	0.97981	-0.20763
B115	0.97995	-0.20642
B116	0.97754	-0.24002
B117	0.97066	-0.33926
B118	0.97995	-0.20642
B119	0.97747	-0.24050
B120	0.97735	-0.23980
B121	0.97744	-0.24083
B122	0.97747	-0.24050
B123	0.97747	-0.24050
B124	0.97754	-0.24002

★Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 2 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.4. See Table 5.12 for the commentary on it.

Table 5.12: Commentary on Case 2 simulation's tree pattern in Figure 5.4[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node (furthest upstream) feeds the grid, which is backup feeder B36. Figure 5.4 has three more smaller green nodes: The one to the left is main feeder B1, the middle one is backup feeder B62, and the one to the right is backup feeder B88.
A battery or an EV	cyan	None in this category.

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	A string of two lines near backup feeder B36.
$40\% < F \leq 60\%$	yellow	A string of twelve lines on the main branch, nearly furthest upstream, only a string of two lines between the feeder and the beforementioned string.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A string of two lines doesn't transmit to backup feeder B36, a string of two lines doesn't transmit to bus B123, a string of two lines doesn't transmit to backup feeder B62, and a string of two lines doesn't transmit to backup feeder B88.

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node (furthest upstream) is the grid's feeder, main feeder B1, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	Approximately half of the grid's 124 nodes are colored orange, all connected furthest downstream of the grid (coloring the backup feeders B62 and B88 orange, both connected to violet lines).
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

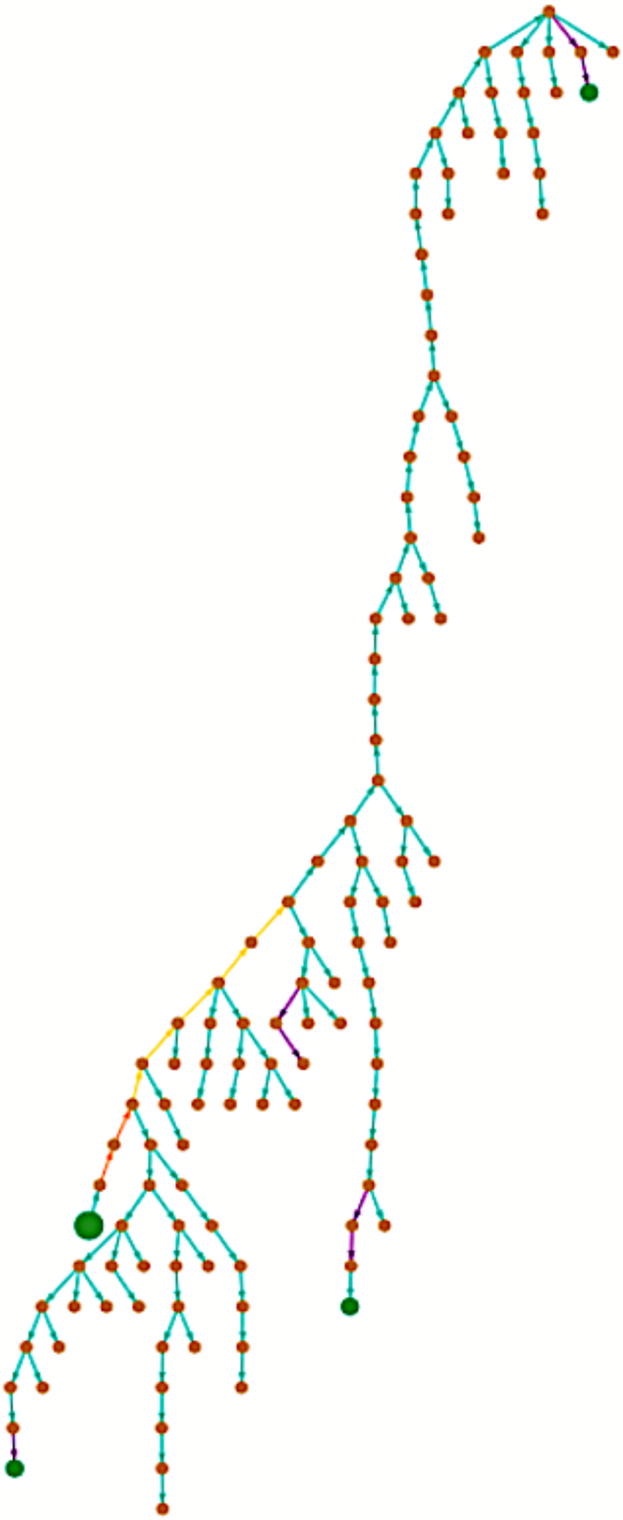


Figure 5.4: Case 2 simulation's tree pattern★
Backup feeder B36 as feeder.

★See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 3 simulation's line flows

This simulation's line flows are seen in Tables 5.14 and 5.15. See Table 5.13 for the commentary on them.

Table 5.13: Commentary on Case 3 simulation's line flows in Tables 5.14 and 5.15[★]

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	L95, thus overloaded.
$80\% < F \leq 100\%$	pink	L88 and L94.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	L86 and L96 (L97 is connected to backup feeder B62, feeding the grid).
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	L1 (connected to main feeder B1), L65 and L66 (connected to backup feeder B36), L82, L120 and L121 (connected to backup feeder B88).

Line flow range	Color	Line(s) or line flow value [pu]
Min. active flow in line(s)	violet	L1, L65, L66, L82, L120 and L121.
Min. active flow in line(s)	seagreen	L81 (connected to line L82), probably not exactly zero, thus not tagged violet.
Min. active line flow	violet	0.0 pu
Max. active flow in line	seagreen	L97 (connected to backup feeder B62, feeding the grid).
Max. active line flow	seagreen	0.6530 pu

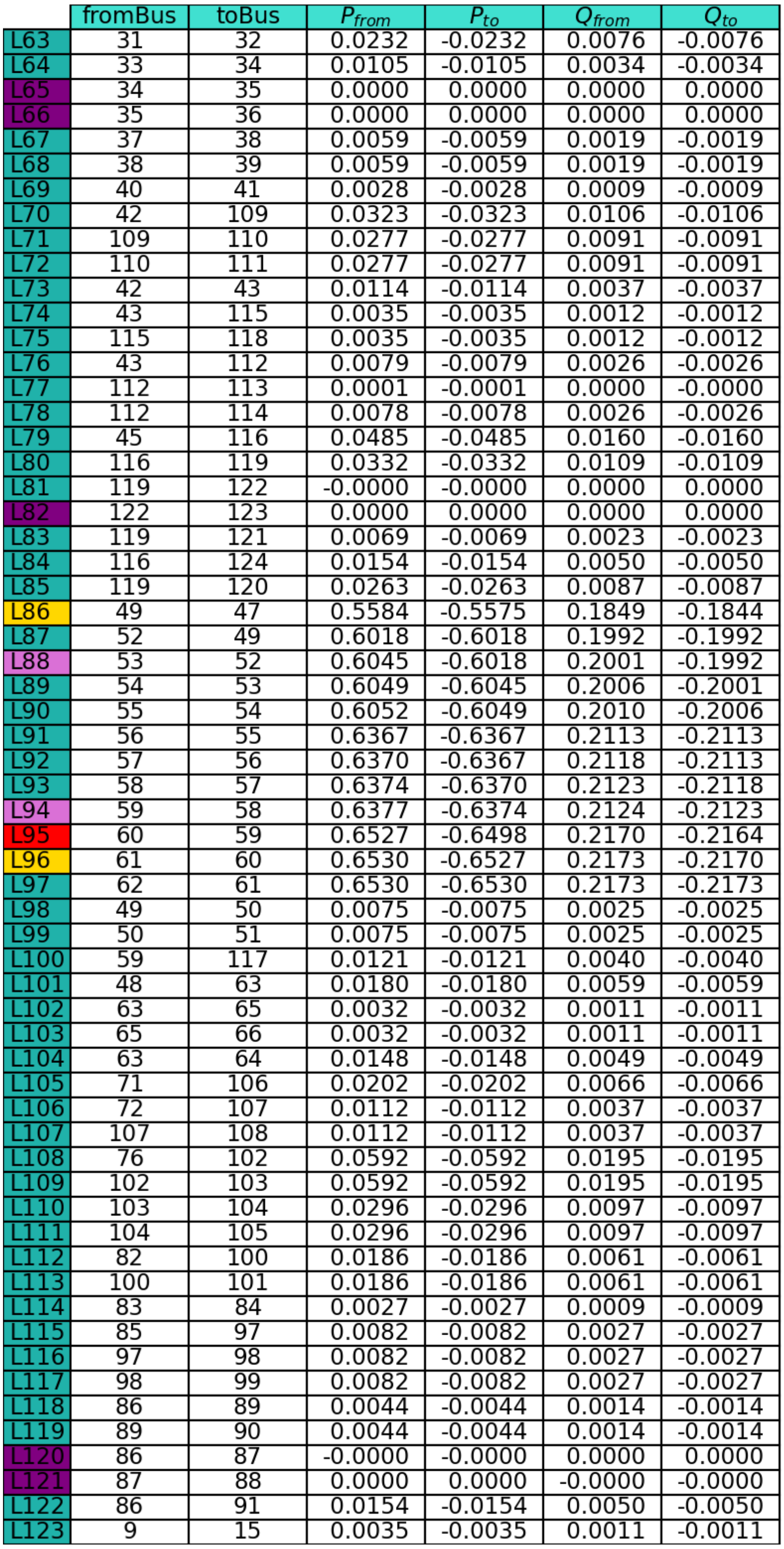
[★] F is a percentage of a line flow divided by its line's capacity. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Table 5.14: Case 3 simulation's line flows for lines L1-L62[★]

	fromBus	toBus	P_{from}	P_{to}	Q_{from}	Q_{to}
L1	2	1	0.0000	0.0000	0.0000	0.0000
L2	3	2	0.0103	-0.0103	0.0034	-0.0034
L3	4	3	0.0103	-0.0103	0.0034	-0.0034
L4	5	4	0.0415	-0.0415	0.0136	-0.0136
L5	7	5	0.0485	-0.0485	0.0159	-0.0159
L6	9	7	0.0617	-0.0617	0.0203	-0.0203
L7	12	9	0.0767	-0.0766	0.0252	-0.0252
L8	26	12	0.1040	-0.1040	0.0342	-0.0342
L9	33	26	0.1458	-0.1457	0.0480	-0.0479
L10	37	33	0.1564	-0.1563	0.0515	-0.0514
L11	40	37	0.1625	-0.1622	0.0536	-0.0534
L12	42	40	0.1713	-0.1710	0.0566	-0.0564
L13	44	42	0.2152	-0.2150	0.0711	-0.0709
L14	45	44	0.2184	-0.2183	0.0721	-0.0721
L15	46	45	0.2673	-0.2669	0.0883	-0.0881
L16	47	46	0.2973	-0.2964	0.0985	-0.0979
L17	47	48	0.2466	-0.2466	0.0815	-0.0814
L18	48	67	0.2286	-0.2284	0.0755	-0.0754
L19	67	68	0.2259	-0.2256	0.0746	-0.0744
L20	68	69	0.2247	-0.2244	0.0741	-0.0739
L21	69	70	0.2224	-0.2223	0.0733	-0.0731
L22	70	71	0.2137	-0.2136	0.0703	-0.0703
L23	71	72	0.1934	-0.1933	0.0636	-0.0635
L24	72	73	0.1821	-0.1821	0.0599	-0.0599
L25	73	74	0.1366	-0.1366	0.0449	-0.0449
L26	74	75	0.1366	-0.1365	0.0449	-0.0449
L27	75	76	0.1365	-0.1365	0.0449	-0.0449
L28	76	77	0.0773	-0.0773	0.0254	-0.0254
L29	77	78	0.0773	-0.0773	0.0254	-0.0254
L30	78	79	0.0773	-0.0772	0.0254	-0.0254
L31	79	80	0.0772	-0.0772	0.0254	-0.0254
L32	80	81	0.0751	-0.0751	0.0247	-0.0247
L33	81	82	0.0751	-0.0750	0.0247	-0.0247
L34	82	83	0.0564	-0.0564	0.0186	-0.0186
L35	83	85	0.0537	-0.0537	0.0177	-0.0177
L36	85	86	0.0455	-0.0455	0.0150	-0.0150
L37	86	92	0.0258	-0.0258	0.0085	-0.0085
L38	92	93	0.0184	-0.0184	0.0060	-0.0060
L39	93	94	0.0184	-0.0184	0.0060	-0.0060
L40	94	95	0.0184	-0.0184	0.0060	-0.0060
L41	95	96	0.0184	-0.0184	0.0060	-0.0060
L42	4	6	0.0312	-0.0312	0.0103	-0.0103
L43	5	8	0.0070	-0.0070	0.0023	-0.0023
L44	7	10	0.0077	-0.0077	0.0025	-0.0025
L45	7	11	0.0054	-0.0054	0.0018	-0.0018
L46	9	13	0.0114	-0.0114	0.0038	-0.0038
L47	13	14	0.0045	-0.0045	0.0015	-0.0015
L48	12	16	0.0273	-0.0273	0.0090	-0.0090
L49	16	18	0.0189	-0.0189	0.0062	-0.0062
L50	18	19	0.0130	-0.0130	0.0043	-0.0043
L51	19	21	0.0031	-0.0031	0.0010	-0.0010
L52	21	22	0.0015	-0.0015	0.0005	-0.0005
L53	22	23	0.0015	-0.0015	0.0005	-0.0005
L54	23	24	0.0015	-0.0015	0.0005	-0.0005
L55	24	25	0.0015	-0.0015	0.0005	-0.0005
L56	16	17	0.0084	-0.0084	0.0028	-0.0028
L57	19	20	0.0099	-0.0099	0.0032	-0.0032
L58	26	27	0.0417	-0.0417	0.0137	-0.0137
L59	27	28	0.0417	-0.0417	0.0137	-0.0137
L60	28	29	0.0349	-0.0349	0.0115	-0.0115
L61	29	30	0.0349	-0.0349	0.0115	-0.0115
L62	30	31	0.0349	-0.0349	0.0115	-0.0115

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Table 5.15: Case 3 simulation's line flows for lines L63-L123★



	fromBus	toBus	P_{from}	P_{to}	Q_{from}	Q_{to}
L63	31	32	0.0232	-0.0232	0.0076	-0.0076
L64	33	34	0.0105	-0.0105	0.0034	-0.0034
L65	34	35	0.0000	0.0000	0.0000	0.0000
L66	35	36	0.0000	0.0000	0.0000	0.0000
L67	37	38	0.0059	-0.0059	0.0019	-0.0019
L68	38	39	0.0059	-0.0059	0.0019	-0.0019
L69	40	41	0.0028	-0.0028	0.0009	-0.0009
L70	42	109	0.0323	-0.0323	0.0106	-0.0106
L71	109	110	0.0277	-0.0277	0.0091	-0.0091
L72	110	111	0.0277	-0.0277	0.0091	-0.0091
L73	42	43	0.0114	-0.0114	0.0037	-0.0037
L74	43	115	0.0035	-0.0035	0.0012	-0.0012
L75	115	118	0.0035	-0.0035	0.0012	-0.0012
L76	43	112	0.0079	-0.0079	0.0026	-0.0026
L77	112	113	0.0001	-0.0001	0.0000	-0.0000
L78	112	114	0.0078	-0.0078	0.0026	-0.0026
L79	45	116	0.0485	-0.0485	0.0160	-0.0160
L80	116	119	0.0332	-0.0332	0.0109	-0.0109
L81	119	122	-0.0000	-0.0000	0.0000	0.0000
L82	122	123	0.0000	0.0000	0.0000	0.0000
L83	119	121	0.0069	-0.0069	0.0023	-0.0023
L84	116	124	0.0154	-0.0154	0.0050	-0.0050
L85	119	120	0.0263	-0.0263	0.0087	-0.0087
L86	49	47	0.5584	-0.5575	0.1849	-0.1844
L87	52	49	0.6018	-0.6018	0.1992	-0.1992
L88	53	52	0.6045	-0.6018	0.2001	-0.1992
L89	54	53	0.6049	-0.6045	0.2006	-0.2001
L90	55	54	0.6052	-0.6049	0.2010	-0.2006
L91	56	55	0.6367	-0.6367	0.2113	-0.2113
L92	57	56	0.6370	-0.6367	0.2118	-0.2113
L93	58	57	0.6374	-0.6370	0.2123	-0.2118
L94	59	58	0.6377	-0.6374	0.2124	-0.2123
L95	60	59	0.6527	-0.6498	0.2170	-0.2164
L96	61	60	0.6530	-0.6527	0.2173	-0.2170
L97	62	61	0.6530	-0.6530	0.2173	-0.2173
L98	49	50	0.0075	-0.0075	0.0025	-0.0025
L99	50	51	0.0075	-0.0075	0.0025	-0.0025
L100	59	117	0.0121	-0.0121	0.0040	-0.0040
L101	48	63	0.0180	-0.0180	0.0059	-0.0059
L102	63	65	0.0032	-0.0032	0.0011	-0.0011
L103	65	66	0.0032	-0.0032	0.0011	-0.0011
L104	63	64	0.0148	-0.0148	0.0049	-0.0049
L105	71	106	0.0202	-0.0202	0.0066	-0.0066
L106	72	107	0.0112	-0.0112	0.0037	-0.0037
L107	107	108	0.0112	-0.0112	0.0037	-0.0037
L108	76	102	0.0592	-0.0592	0.0195	-0.0195
L109	102	103	0.0592	-0.0592	0.0195	-0.0195
L110	103	104	0.0296	-0.0296	0.0097	-0.0097
L111	104	105	0.0296	-0.0296	0.0097	-0.0097
L112	82	100	0.0186	-0.0186	0.0061	-0.0061
L113	100	101	0.0186	-0.0186	0.0061	-0.0061
L114	83	84	0.0027	-0.0027	0.0009	-0.0009
L115	85	97	0.0082	-0.0082	0.0027	-0.0027
L116	97	98	0.0082	-0.0082	0.0027	-0.0027
L117	98	99	0.0082	-0.0082	0.0027	-0.0027
L118	86	89	0.0044	-0.0044	0.0014	-0.0014
L119	89	90	0.0044	-0.0044	0.0014	-0.0014
L120	86	87	-0.0000	-0.0000	0.0000	0.0000
L121	87	88	0.0000	0.0000	-0.0000	-0.0000
L122	86	91	0.0154	-0.0154	0.0050	-0.0050
L123	9	15	0.0035	-0.0035	0.0011	-0.0011

★Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 3 simulation's bus voltages

This simulation's bus voltages are seen in Table 5.17. See Table 5.16 for the commentary on them.

Table 5.16: Commentary on Case 3 simulation's bus voltages in Table 5.17[★]

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category, but bus B61 (one line downstream of backup feeder B62, feeding the grid) should be colored pink as it has a voltage magnitude of 1.0 pu.
$V = 1.0$ pu	green	Backup feeder B62 feeds the grid.
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	None in this category.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category, but the buses main feeder B1, B35, backup feeder B36, B87, backup feeder B88, B122 and B123 should be violet, since no power is transmitted to them (Tables 5.14 and 5.15).

Bus voltage range	Color	Bus(es) or bus voltage magnitude [pu]
Min. voltage at bus	brown	Main feeder B1, B2 and B6, thus they are the buses furthest downstream.
Min. voltage magnitude	brown	0.97313 pu
Max. voltage at bus	green	Backup feeder B62, feeding the grid.
Max. voltage magnitude	green	1.0 pu

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Table 5.17: *Case 3 simulation's bus voltages*[★]

	V_{mag}	Θ_V
B1	0.97313	-0.31304
B2	0.97313	-0.31304
B3	0.97314	-0.31233
B4	0.97317	-0.31168
B5	0.97320	-0.31099
B6	0.97313	-0.31211
B7	0.97342	-0.30775
B8	0.97320	-0.31099
B9	0.97358	-0.30544
B10	0.97342	-0.30775
B11	0.97342	-0.30775
B12	0.97418	-0.29664
B13	0.97356	-0.30548
B14	0.97354	-0.30552
B15	0.97358	-0.30544
B16	0.97415	-0.29684
B17	0.97415	-0.29685
B18	0.97399	-0.29795
B19	0.97386	-0.29877
B20	0.97386	-0.29877
B21	0.97384	-0.29877
B22	0.97384	-0.29877
B23	0.97383	-0.29867
B24	0.97381	-0.29842
B25	0.97381	-0.29842
B26	0.97484	-0.28700
B27	0.97482	-0.28790
B28	0.97482	-0.28791
B29	0.97482	-0.28791
B30	0.97480	-0.28778
B31	0.97479	-0.28771
B32	0.97476	-0.28884
B33	0.97572	-0.27424
B34	0.97563	-0.27486
B35	0.97563	-0.27486
B36	0.97563	-0.27486
B37	0.97640	-0.26441
B38	0.97638	-0.26455
B39	0.97638	-0.26455
B40	0.97787	-0.24308
B41	0.97785	-0.24297
B42	0.97972	-0.21629
B43	0.97972	-0.21647
B44	0.98071	-0.20205
B45	0.98089	-0.19956
B46	0.98257	-0.17529
B47	0.98583	-0.12861
B48	0.98555	-0.13257
B49	0.98747	-0.10539
B50	0.98742	-0.10510
B51	0.98742	-0.10510
B52	0.98747	-0.10538
B53	0.99197	-0.10616
B54	0.99270	-0.07443
B55	0.99332	-0.04796
B56	0.99332	-0.04795
B57	0.99393	-0.02135
B58	0.99473	0.01303
B59	0.99514	0.01292
B60	0.99947	-0.01362
B61	1.00000	-0.00001
B62	1.00000	0.00000
B63	0.98512	-0.13182
B64	0.98512	-0.13183
B65	0.98508	-0.13210
B66	0.98508	-0.13210
B67	0.98462	-0.14594
B68	0.98340	-0.16348
B69	0.98181	-0.18637
B70	0.98100	-0.19811
B71	0.98049	-0.20554
B72	0.97972	-0.21662
B73	0.97971	-0.21676
B74	0.97970	-0.21691
B75	0.97914	-0.21825
B76	0.97914	-0.21826
B77	0.97914	-0.21826
B78	0.97914	-0.21826
B79	0.97862	-0.21953
B80	0.97861	-0.21957
B81	0.97861	-0.21962
B82	0.97854	-0.22062
B83	0.97822	-0.22525
B84	0.97822	-0.22525
B85	0.97798	-0.22872
B86	0.97778	-0.23159
B87	0.97778	-0.23159
B88	0.97778	-0.23159
B89	0.97778	-0.23166
B90	0.97778	-0.23169
B91	0.97777	-0.23210
B92	0.97778	-0.23159
B93	0.97778	-0.23160
B94	0.97773	-0.23298
B95	0.97772	-0.23300
B96	0.97765	-0.23256
B97	0.97790	-0.22847
B98	0.97790	-0.22855
B99	0.97790	-0.22855
B100	0.97846	-0.22014
B101	0.97846	-0.22014
B102	0.97914	-0.21826
B103	0.97910	-0.21836
B104	0.97907	-0.21845
B105	0.97907	-0.21845
B106	0.98046	-0.20539
B107	0.97968	-0.21728
B108	0.97968	-0.21729
B109	0.97959	-0.21719
B110	0.97946	-0.21804
B111	0.97946	-0.21806
B112	0.97968	-0.21720
B113	0.97968	-0.21720
B114	0.97957	-0.21800
B115	0.97970	-0.21679
B116	0.98077	-0.20031
B117	0.99514	0.01292
B118	0.97970	-0.21679
B119	0.98070	-0.20078
B120	0.98059	-0.20009
B121	0.98068	-0.20112
B122	0.98070	-0.20078
B123	0.98070	-0.20078
B124	0.98077	-0.20031

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 3 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.6. See Table 5.18 for the commentary on it.

Table 5.18: Commentary on Case 3 simulation's tree pattern in Figure 5.6[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the grid, which is backup feeder B62. The three other smaller green nodes are the other alternative feeders: The left one is main feeder B1. The middle one is backup feeder B36. The right one is backup feeder B88.
A battery or an EV	cyan	None in this category.
Line flow [%]	**Color**	**Categorized lines**
$F > 100\%$	red	The third line from the feeder, thus overloaded.
$80\% < F \leq 100\%$	pink	The fourth and tenth line from the feeder.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	The second and twelfth line from the feeder.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A line doesn't transmit to main feeder B1, a string of two lines doesn't transmit to backup feeder B36, a line doesn't transmit to bus B123, and a string of two lines doesn't transmit to backup feeder B88.
Bus voltage [pu]	**Color**	**Categorized buses**
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the grid's feeder, backup feeder B62, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	None in this category.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

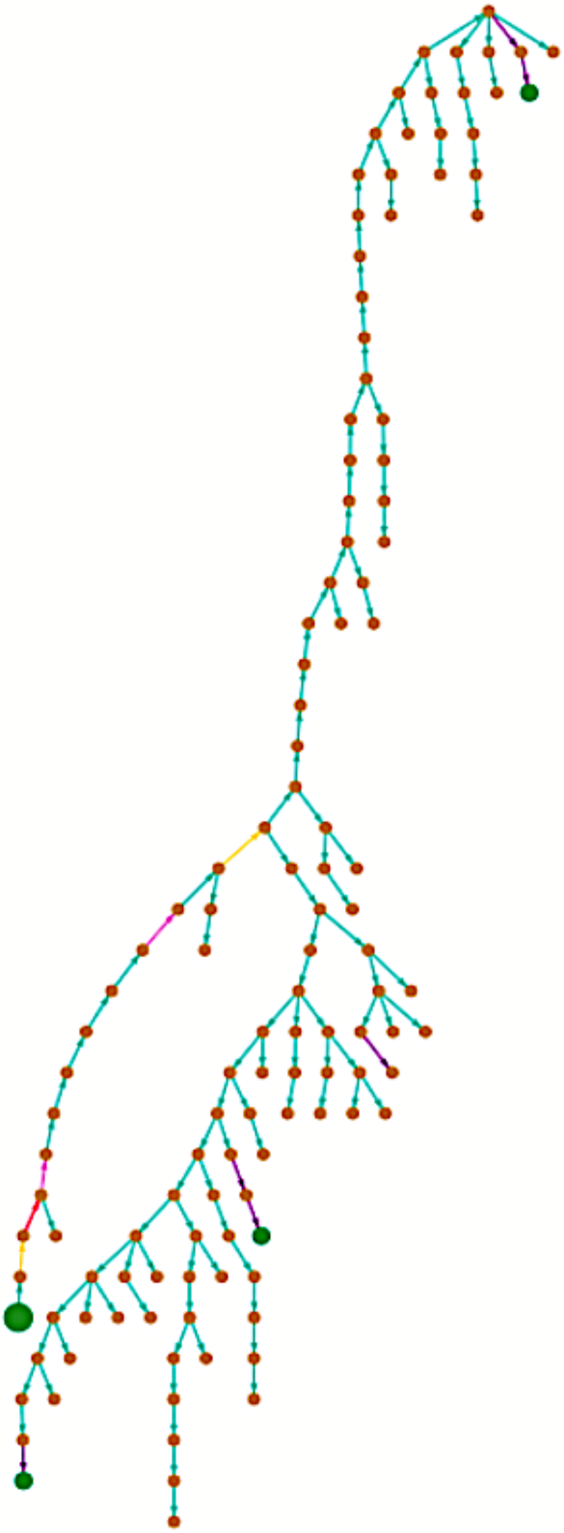


Figure 5.6: Case 3 simulation's tree pattern[★]
Backup feeder B62 as feeder.

[★] See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 4 simulation's line flows

This simulation's line flows are seen in Tables 5.20 and 5.21. See Table 5.19 for the commentary on them.

Table 5.19: Commentary on Case 4 simulation's line flows in Tables 5.20 and 5.21[★]

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	L30, L120 and L121 (connected to backup feeder B88, feeding the grid).
$40\% < F \leq 60\%$	yellow	L18-L26 and L31-L36, where L36 (connected to bus B86) is two lines apart from backup feeder B88.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	L1 (connected to main feeder B1), L66 (connected to backup feeder B36), L81 and L82, L95 and L96 (L97 is connected to backup feeder B62).

Line flow range	Color	Line(s) or line flow value [pu]
Min. active flow in line(s)	violet	L1, L66, L81, L82, L95 and L96.
Min. active flow in line(s)	seagreen	L65 and L97, the lines should be colored violet.
Min. active line flow	violet	0.0 pu

Max. active flow in line	orange	L121, connected to backup feeder B88, feeding the grid.
Max. active line flow	orange	0.6580 pu

[★] F is a percentage of a line flow divided by its line's capacity. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Table 5.20: Case 4 simulation's line flows for lines L1-L62[★]

	fromBus	toBus	P_{from}	P_{to}	Q_{from}	Q_{to}
L1	2	1	0.0000	0.0000	0.0000	0.0000
L2	3	2	0.0103	-0.0103	0.0034	-0.0034
L3	4	3	0.0103	-0.0103	0.0034	-0.0034
L4	5	4	0.0415	-0.0415	0.0136	-0.0136
L5	7	5	0.0485	-0.0485	0.0159	-0.0159
L6	9	7	0.0617	-0.0617	0.0203	-0.0203
L7	12	9	0.0767	-0.0766	0.0252	-0.0252
L8	26	12	0.1040	-0.1040	0.0342	-0.0342
L9	33	26	0.1458	-0.1457	0.0480	-0.0479
L10	37	33	0.1564	-0.1563	0.0515	-0.0514
L11	40	37	0.1625	-0.1623	0.0536	-0.0534
L12	42	40	0.1714	-0.1711	0.0566	-0.0564
L13	44	42	0.2152	-0.2150	0.0711	-0.0710
L14	45	44	0.2184	-0.2184	0.0721	-0.0721
L15	46	45	0.2674	-0.2669	0.0884	-0.0881
L16	47	46	0.2974	-0.2965	0.0985	-0.0979
L17	48	47	0.3983	-0.3981	0.1317	-0.1316
L18	67	48	0.4170	-0.4163	0.1381	-0.1376
L19	68	67	0.4204	-0.4195	0.1395	-0.1389
L20	69	68	0.4225	-0.4213	0.1405	-0.1397
L21	70	69	0.4250	-0.4244	0.1415	-0.1411
L22	71	70	0.4340	-0.4336	0.1446	-0.1443
L23	72	71	0.4550	-0.4542	0.1517	-0.1512
L24	73	72	0.4661	-0.4661	0.1554	-0.1554
L25	74	73	0.5117	-0.5117	0.1704	-0.1704
L26	75	74	0.5128	-0.5117	0.1708	-0.1704
L27	76	75	0.5128	-0.5128	0.1708	-0.1708
L28	77	76	0.5720	-0.5720	0.1902	-0.1902
L29	78	77	0.5720	-0.5720	0.1902	-0.1902
L30	79	78	0.5742	-0.5720	0.1911	-0.1902
L31	80	79	0.5742	-0.5742	0.1911	-0.1911
L32	81	80	0.5764	-0.5764	0.1918	-0.1918
L33	82	81	0.5767	-0.5764	0.1920	-0.1918
L34	83	82	0.5972	-0.5953	0.1993	-0.1981
L35	85	83	0.6013	-0.5999	0.2011	-0.2002
L36	86	85	0.6110	-0.6096	0.2047	-0.2038
L37	86	92	0.0258	-0.0258	0.0085	-0.0085
L38	92	93	0.0184	-0.0184	0.0060	-0.0060
L39	93	94	0.0184	-0.0184	0.0060	-0.0060
L40	94	95	0.0184	-0.0184	0.0060	-0.0060
L41	95	96	0.0184	-0.0184	0.0060	-0.0060
L42	4	6	0.0312	-0.0312	0.0103	-0.0103
L43	5	8	0.0070	-0.0070	0.0023	-0.0023
L44	7	10	0.0077	-0.0077	0.0025	-0.0025
L45	7	11	0.0054	-0.0054	0.0018	-0.0018
L46	9	13	0.0114	-0.0114	0.0038	-0.0038
L47	13	14	0.0045	-0.0045	0.0015	-0.0015
L48	12	16	0.0273	-0.0273	0.0090	-0.0090
L49	16	18	0.0189	-0.0189	0.0062	-0.0062
L50	18	19	0.0130	-0.0130	0.0043	-0.0043
L51	19	21	0.0031	-0.0031	0.0010	-0.0010
L52	21	22	0.0015	-0.0015	0.0005	-0.0005
L53	22	23	0.0015	-0.0015	0.0005	-0.0005
L54	23	24	0.0015	-0.0015	0.0005	-0.0005
L55	24	25	0.0015	-0.0015	0.0005	-0.0005
L56	16	17	0.0084	-0.0084	0.0028	-0.0028
L57	19	20	0.0099	-0.0099	0.0032	-0.0032
L58	26	27	0.0417	-0.0417	0.0137	-0.0137
L59	27	28	0.0417	-0.0417	0.0137	-0.0137
L60	28	29	0.0349	-0.0349	0.0115	-0.0115
L61	29	30	0.0349	-0.0349	0.0115	-0.0115
L62	30	31	0.0349	-0.0349	0.0115	-0.0115

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Table 5.21: Case 4 simulation's line flows for lines L63-L123[★]

	fromBus	toBus	P_{from}	P_{to}	Q_{from}	Q_{to}
L63	31	32	0.0232	-0.0232	0.0076	-0.0076
L64	33	34	0.0105	-0.0105	0.0034	-0.0034
L65	34	35	0.0000	0.0000	0.0000	0.0000
L66	35	36	0.0000	0.0000	0.0000	0.0000
L67	37	38	0.0059	-0.0059	0.0019	-0.0019
L68	38	39	0.0059	-0.0059	0.0019	-0.0019
L69	40	41	0.0028	-0.0028	0.0009	-0.0009
L70	42	109	0.0323	-0.0323	0.0106	-0.0106
L71	109	110	0.0277	-0.0277	0.0091	-0.0091
L72	110	111	0.0277	-0.0277	0.0091	-0.0091
L73	42	43	0.0114	-0.0114	0.0037	-0.0037
L74	43	115	0.0035	-0.0035	0.0012	-0.0012
L75	115	118	0.0035	-0.0035	0.0012	-0.0012
L76	43	112	0.0079	-0.0079	0.0026	-0.0026
L77	112	113	0.0001	-0.0001	0.0000	-0.0000
L78	112	114	0.0078	-0.0078	0.0026	-0.0026
L79	45	116	0.0485	-0.0485	0.0160	-0.0160
L80	116	119	0.0332	-0.0332	0.0109	-0.0109
L81	119	122	0.0000	0.0000	-0.0000	-0.0000
L82	122	123	-0.0000	-0.0000	0.0000	0.0000
L83	119	121	0.0069	-0.0069	0.0023	-0.0023
L84	116	124	0.0154	-0.0154	0.0050	-0.0050
L85	119	120	0.0263	-0.0263	0.0087	-0.0087
L86	47	49	0.0871	-0.0871	0.0286	-0.0286
L87	49	52	0.0437	-0.0437	0.0144	-0.0144
L88	52	53	0.0437	-0.0437	0.0144	-0.0144
L89	53	54	0.0437	-0.0437	0.0144	-0.0144
L90	54	55	0.0437	-0.0437	0.0144	-0.0144
L91	55	56	0.0121	-0.0121	0.0040	-0.0040
L92	56	57	0.0121	-0.0121	0.0040	-0.0040
L93	57	58	0.0121	-0.0121	0.0040	-0.0040
L94	58	59	0.0121	-0.0121	0.0040	-0.0040
L95	59	60	0.0000	0.0000	-0.0000	-0.0000
L96	60	61	0.0000	0.0000	0.0000	0.0000
L97	61	62	-0.0000	-0.0000	-0.0000	-0.0000
L98	49	50	0.0075	-0.0075	0.0025	-0.0025
L99	50	51	0.0075	-0.0075	0.0025	-0.0025
L100	59	117	0.0121	-0.0121	0.0040	-0.0040
L101	48	63	0.0180	-0.0180	0.0059	-0.0059
L102	63	65	0.0032	-0.0032	0.0011	-0.0011
L103	65	66	0.0032	-0.0032	0.0011	-0.0011
L104	63	64	0.0148	-0.0148	0.0049	-0.0049
L105	71	106	0.0202	-0.0202	0.0066	-0.0066
L106	72	107	0.0112	-0.0112	0.0037	-0.0037
L107	107	108	0.0112	-0.0112	0.0037	-0.0037
L108	76	102	0.0592	-0.0592	0.0195	-0.0195
L109	102	103	0.0592	-0.0592	0.0195	-0.0195
L110	103	104	0.0296	-0.0296	0.0097	-0.0097
L111	104	105	0.0296	-0.0296	0.0097	-0.0097
L112	82	100	0.0186	-0.0186	0.0061	-0.0061
L113	100	101	0.0186	-0.0186	0.0061	-0.0061
L114	83	84	0.0027	-0.0027	0.0009	-0.0009
L115	85	97	0.0082	-0.0082	0.0027	-0.0027
L116	97	98	0.0082	-0.0082	0.0027	-0.0027
L117	98	99	0.0082	-0.0082	0.0027	-0.0027
L118	86	89	0.0044	-0.0044	0.0014	-0.0014
L119	89	90	0.0044	-0.0044	0.0014	-0.0014
L120	87	86	0.6579	-0.6565	0.2206	-0.2197
L121	88	87	0.6580	-0.6579	0.2206	-0.2206
L122	86	91	0.0154	-0.0154	0.0050	-0.0050
L123	9	15	0.0035	-0.0035	0.0011	-0.0011

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 4 simulation's bus voltages

This simulation's bus voltages are seen in Table 5.23. See Table 5.22 for the commentary on them.

Table 5.22: Commentary on Case 4 simulation's bus voltages in Table 5.23[★]

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V <$ 1.1 pu	pink	None in this category.
$V =$ 1.0 pu	green	Backup feeder B88 feeds the grid.
0.96 pu $< V <$ 1.0 pu	brown	The rest of the buses.
0.94 pu $< V \leq$ 0.96 pu	orange	Main feeder B1-B32
0.0 pu $< V \leq$ 0.94 pu	yellow	None in this category.
Zero node potential	violet	None in this category, but the buses main feeder B1, B35, backup feeder B36, B60, B61, backup feeder B62, B122 and B123 should be violet, since no power is transmitted to them (Tables 5.20 and 5.21).

Bus voltage range	Color	Bus(es) or bus voltage magnitude [pu]
Min. voltage at bus	orange	Main feeder B1 and B2
Min. voltage magnitude	orange	0.95756 pu
Max. voltage at bus	green	Backup feeder B88, feeding the grid.
Max. voltage magnitude	green	1.0 pu

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Table 5.23: Case 4 simulation's bus voltages[★]

	V_{mag}	Θ_V
B1	0.95756	-0.53720
B2	0.95756	-0.53720
B3	0.95758	-0.53647
B4	0.95760	-0.53580
B5	0.95763	-0.53509
B6	0.95757	-0.53624
B7	0.95786	-0.53174
B8	0.95763	-0.53509
B9	0.95802	-0.52935
B10	0.95786	-0.53174
B11	0.95786	-0.53174
B12	0.95863	-0.52027
B13	0.95800	-0.52940
B14	0.95798	-0.52944
B15	0.95802	-0.52935
B16	0.95860	-0.52047
B17	0.95860	-0.52048
B18	0.95844	-0.52161
B19	0.95831	-0.52247
B20	0.95831	-0.52247
B21	0.95828	-0.52247
B22	0.95828	-0.52247
B23	0.95828	-0.52237
B24	0.95826	-0.52210
B25	0.95826	-0.52210
B26	0.95931	-0.51031
B27	0.95928	-0.51124
B28	0.95928	-0.51125
B29	0.95928	-0.51125
B30	0.95926	-0.51111
B31	0.95925	-0.51104
B32	0.95923	-0.51222
B33	0.96020	-0.49714
B34	0.96010	-0.49777
B35	0.96010	-0.49777
B36	0.96010	-0.49777
B37	0.96088	-0.48699
B38	0.96086	-0.48713
B39	0.96086	-0.48713
B40	0.96238	-0.46497
B41	0.96236	-0.46485
B42	0.96427	-0.43730
B43	0.96426	-0.43749
B44	0.96527	-0.42261
B45	0.96545	-0.42003
B46	0.96716	-0.39498
B47	0.97047	-0.34681
B48	0.97093	-0.34022
B49	0.97021	-0.35058
B50	0.97016	-0.35028
B51	0.97016	-0.35028
B52	0.97021	-0.35059
B53	0.96988	-0.35056
B54	0.96983	-0.35297
B55	0.96978	-0.35498
B56	0.96978	-0.35498
B57	0.96977	-0.35551
B58	0.96975	-0.35620
B59	0.96974	-0.35620
B60	0.96974	-0.35620
B61	0.96974	-0.35620
B62	0.96974	-0.35620
B63	0.97049	-0.33944
B64	0.97049	-0.33945
B65	0.97044	-0.33973
B66	0.97044	-0.33973
B67	0.97265	-0.31522
B68	0.97495	-0.28208
B69	0.97795	-0.23892
B70	0.97951	-0.21665
B71	0.98056	-0.20174
B72	0.98236	-0.17614
B73	0.98238	-0.17579
B74	0.98242	-0.17523
B75	0.98451	-0.17070
B76	0.98451	-0.17067
B77	0.98451	-0.17067
B78	0.98451	-0.17065
B79	0.98841	-0.16213
B80	0.98843	-0.16185
B81	0.98845	-0.16148
B82	0.98898	-0.15410
B83	0.99232	-0.10704
B84	0.99232	-0.10704
B85	0.99495	-0.07016
B86	0.99757	-0.03372
B87	0.99994	-0.00089
B88	1.00000	0.00000
B89	0.99757	-0.03378
B90	0.99756	-0.03381
B91	0.99756	-0.03420
B92	0.99757	-0.03372
B93	0.99757	-0.03372
B94	0.99752	-0.03505
B95	0.99751	-0.03507
B96	0.99744	-0.03465
B97	0.99488	-0.06992
B98	0.99487	-0.07000
B99	0.99487	-0.07000
B100	0.98890	-0.15363
B101	0.98890	-0.15363
B102	0.98451	-0.17067
B103	0.98447	-0.17077
B104	0.98443	-0.17086
B105	0.98443	-0.17086
B106	0.98053	-0.20159
B107	0.98231	-0.17679
B108	0.98231	-0.17680
B109	0.96413	-0.43823
B110	0.96400	-0.43911
B111	0.96400	-0.43913
B112	0.96423	-0.43824
B113	0.96423	-0.43824
B114	0.96411	-0.43907
B115	0.96424	-0.43782
B116	0.96533	-0.42081
B117	0.96974	-0.35620
B118	0.96424	-0.43782
B119	0.96526	-0.42129
B120	0.96514	-0.42058
B121	0.96524	-0.42164
B122	0.96526	-0.42129
B123	0.96526	-0.42129
B124	0.96533	-0.42081

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Appendix A to enable zooming of this tree pattern. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 4 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.8. See Table 5.24 for the commentary on it.

Table 5.24: Commentary on Case 4 simulation's tree pattern in Figure 5.8[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the grid, which is backup feeder B88. Figure 5.8 has two smaller green nodes: The left one is backup feeder B36. The right one is backup feeder B62. The grid's fourth alternative feeder, main feeder B1, is colored orange.
A battery or an EV	cyan	None in this category.
Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	The first and second line from the feeder, and the ninth line from the feeder.
$40\% < F \leq 60\%$	yellow	Two strings on the main branch: The first one strings six lines together, two lines apart from the feeder. The latter one strings nine lines together, twelve lines apart from the feeder, four lines apart from the first string.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A line doesn't transmit to main feeder B1 (colored orange), a line doesn't transmit to backup feeder B36, a string of two lines doesn't transmit to bus B123, and a string of three lines doesn't transmit to backup feeder B62 (the line connected to backup feeder B62 is colored seagreen, but it doesn't transmit (Table 5.21).)
Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the grid's feeder, backup feeder B88, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	Approximately one fourth of the grid's 124 nodes are colored orange, all connected furthest downstream (coloring the main feeder B1 orange, connected to a violet line).
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

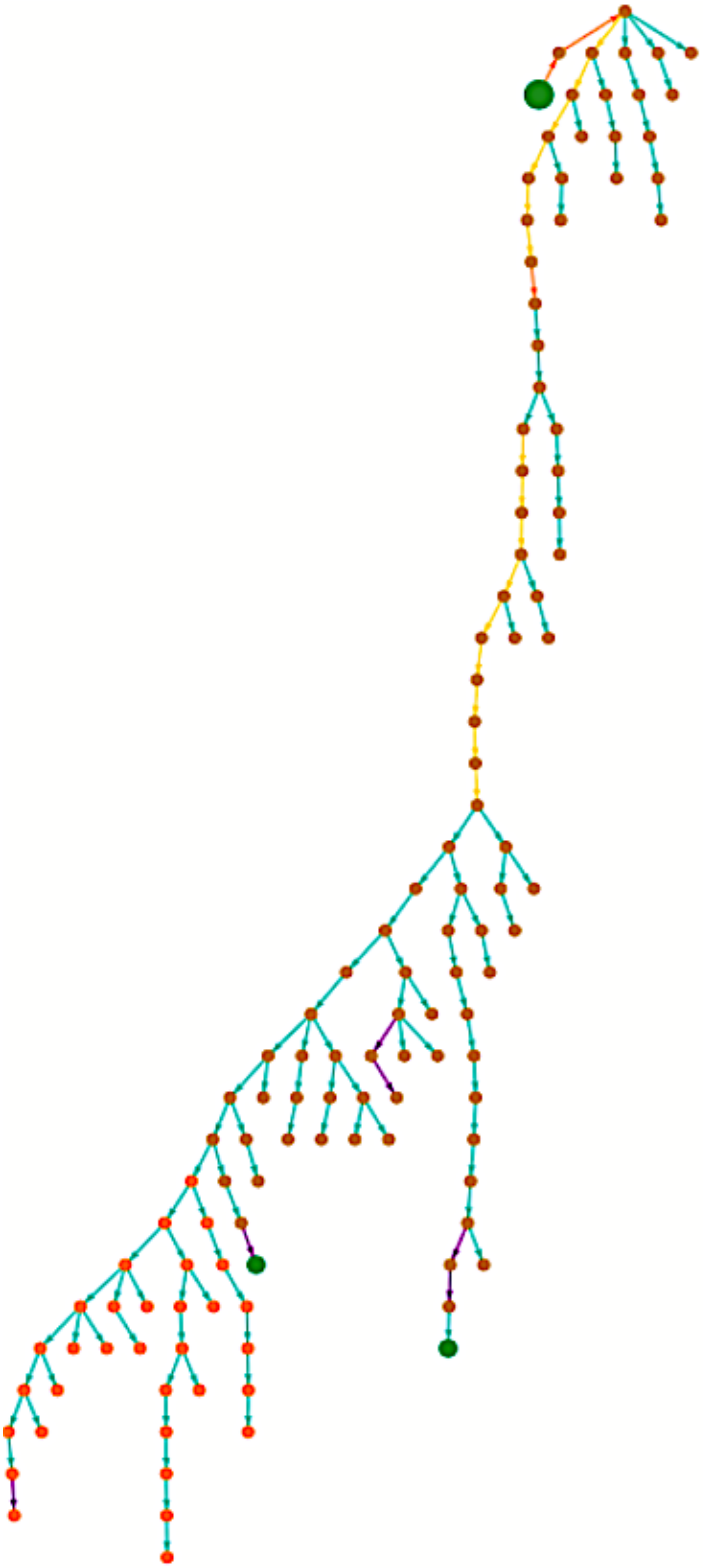


Figure 5.8: Case 4 simulation's tree pattern[★]
Backup feeder B88 as feeder.

[★] Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.1 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

5.2 ... of splitting of the grid

This simulation scenario has four cases, as seen in Table 3.1. Thus four simulations were performed, as seen in Table 5.35, producing four different tree patterns. This scenario's line flows and bus voltages are omitted, downsizing the report.

Case 5 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.9. See Table 5.25 for the commentary on it.

Table 5.25: Commentary on Case 5 simulation's tree pattern in Figure 5.9[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the subgrid, which is main feeder B1. The smaller green node is backup feeder B36. The other two alternative feeders are in the other subgrid.
A battery or an EV	cyan	None in this category.
Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	None in this category.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A string of two lines (L65 and L66) doesn't transmit to backup feeder B36, and a string of two lines (L81 and L82) doesn't transmit to bus B123.
Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the subgrid's feeder, main feeder B1, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	None in this category.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.2a for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

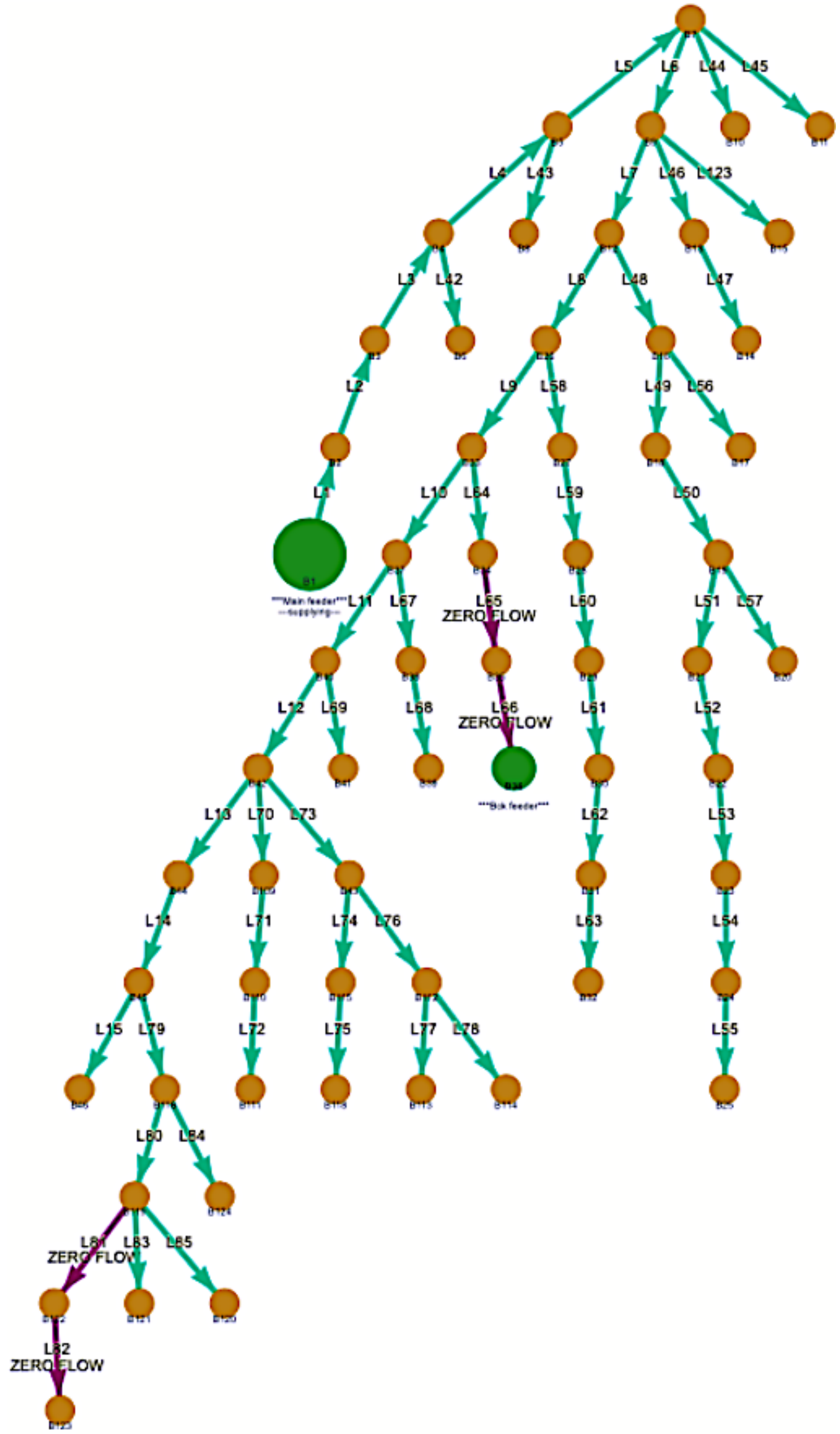


Figure 5.9: Case 5 simulation's tree pattern[★]
Left side subgrid. Main feeder B1 as feeder.

[★] See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.2a for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 6 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.10. See Table 5.26 for the commentary on it.

Table 5.26: Commentary on Case 6 simulation's tree pattern in Figure 5.10[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the subgrid, which is backup feeder B36. The smaller green node is main feeder B1. The other two alternative feeders are in the other subgrid.
A battery or an EV	cyan	None in this category.

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	None in this category.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A line (L1) doesn't transmit to main feeder B1, and a string of two lines (L81 and L82) doesn't transmit to bus B123.

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the subgrid's feeder, backup feeder B36, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	None in this category.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.2a for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

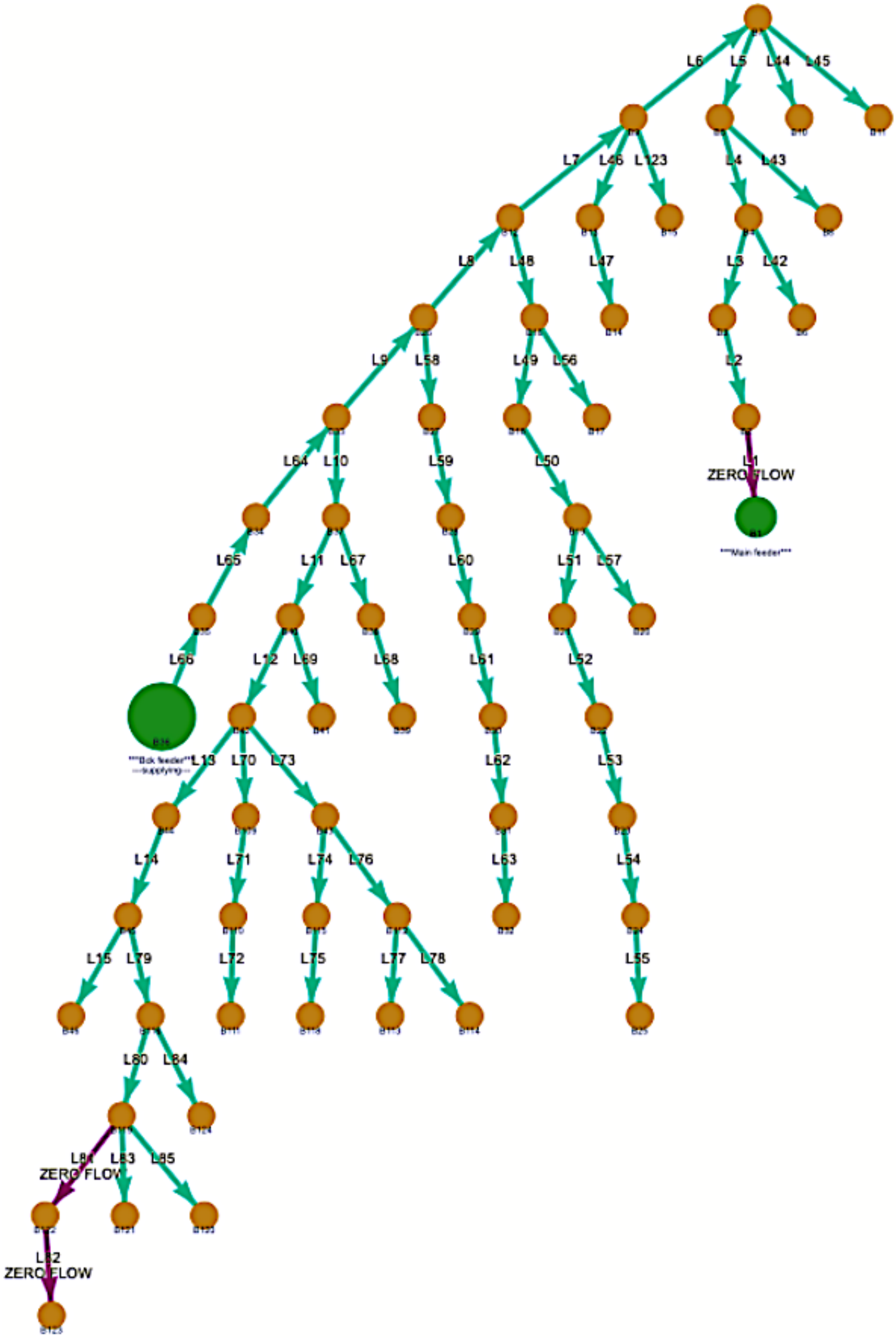


Figure 5.10: Case 6 simulation's tree pattern★
Left side subgrid. Backup feeder B36 as feeder.

★See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.2a for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 7 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.11. See Table 5.27 for the commentary on it.

Table 5.27: Commentary on Case 7 simulation's tree pattern in Figure 5.11[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the subgrid, which is backup feeder B62. The smaller green node is backup feeder B88. The other two alternative feeders are in the other subgrid.
A battery or an EV	cyan	None in this category.
Line flow [%]	**Color**	**Categorized lines**
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	The third, fourth and tenth line from the feeder.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A string of two lines doesn't transmit to backup feeder B88.
Bus voltage [pu]	**Color**	**Categorized buses**
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V <$ 1.1 pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the subgrid's feeder, backup feeder B62, by default set with a voltage of 1.0 pu
0.96 pu $< V <$ 1.0 pu	brown	The rest of the buses.
0.94 pu $< V \leq$ 0.96 pu	orange	None in this category.
0.0 pu $< V \leq$ 0.94 pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.2b for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

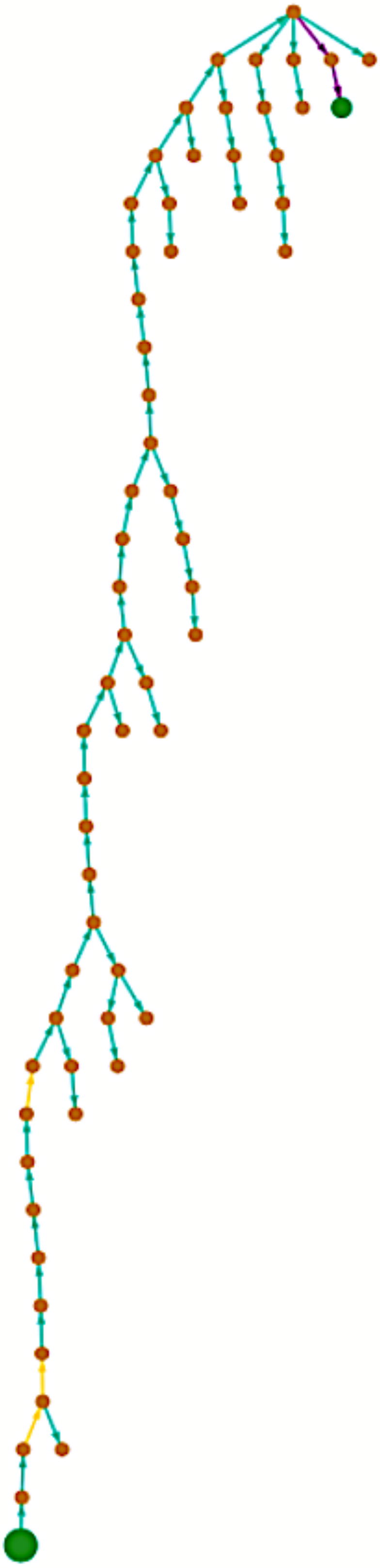


Figure 5.11: Case 7 simulation's tree pattern[★]
Right side subgrid. Backup feeder B62 as feeder.

[★] See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.2b for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 8 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.12. See Table 5.28 for the commentary on it.

Table 5.28: Commentary on Case 8 simulation's tree pattern in Figure 5.12[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the subgrid, which is backup feeder B88. The smaller green node is backup feeder B62, which is furthest downstream. The two other alternative feeders are in the other subgrid.
A battery or an EV	cyan	None in this category.

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	None in this category.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A line connected to backup feeder B62 doesn't transmit. The third line from backup feeder B62 doesn't transmit. Thus the string of three lines connected to backup feeder B62 should all be violet.

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the subgrid's feeder, backup feeder B88, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	None in this category.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.2b for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

Figure 5.12: Case 8 simulation's tree pattern★
Right side subgrid. Backup feeder B88 as feeder.

★See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.2b for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

5.3 ... of local storage as backup feeders

This simulation scenario has four cases, as seen in Table 3.1. Thus four simulations were performed, as seen in Table 5.35, producing four different tree patterns. This scenario's line flows and bus voltages are omitted, downsizing the report.

Case 9 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.13. See Table 5.29 for the commentary on it.

Table 5.29: Commentary on Case 9 simulation's tree pattern in Figure 5.13[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	None of the four alternative feeders are feeding the grid. Three of them are green, while backup feeder B88 furthest downstream is orange. The left green node is main feeder B1. The middle green node is backup feeder B36. The right green node is backup feeder B62.
A battery or an EV	cyan	Four nodes are cyan, representing four identical local storages: Three are charging, as one is discharging, feeding the grid during this feeder outage. The node furthest upstream feeds the grid, cyan bus B5, four lines apart from main feeder B1. The cyan buses further downstream are in chronological order: B115, B70 and B107. Only buses B5 and B70 are on the main branch.
Line flow [%]	**Color**	**Categorized lines**
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	A string of ten lines on the main branch, connected to the discharging battery at cyan bus B5.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	None of the alternative feeders are receiving power. Lines not transmitting: A line to main feeder B1, a string of two lines to backup feeder B36, a string of two lines to bus B123, a string of two lines to bus B61 (connected to backup feeder B62), and a string of two lines to backup feeder B88.
Bus voltage [pu]	**Color**	**Categorized buses**
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V <$ 1.1 pu	pink	None in this category.
$V = 1.0$ pu	cyan	The cyan node furthest upstream is the grid's local storage depleting under this feeder outage, located at bus B5, by default set with a voltage of 1.0 pu
0.96 pu $< V <$ 1.0 pu	brown	The rest of the buses.
0.94 pu $< V \leq$ 0.96 pu	orange	Approximately 30% of the grid's 124 nodes are colored orange, all connected furthest downstream of the grid (coloring the backup feeder B88 orange, connected to a violet string).
0.0 pu $< V \leq$ 0.94 pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.4 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

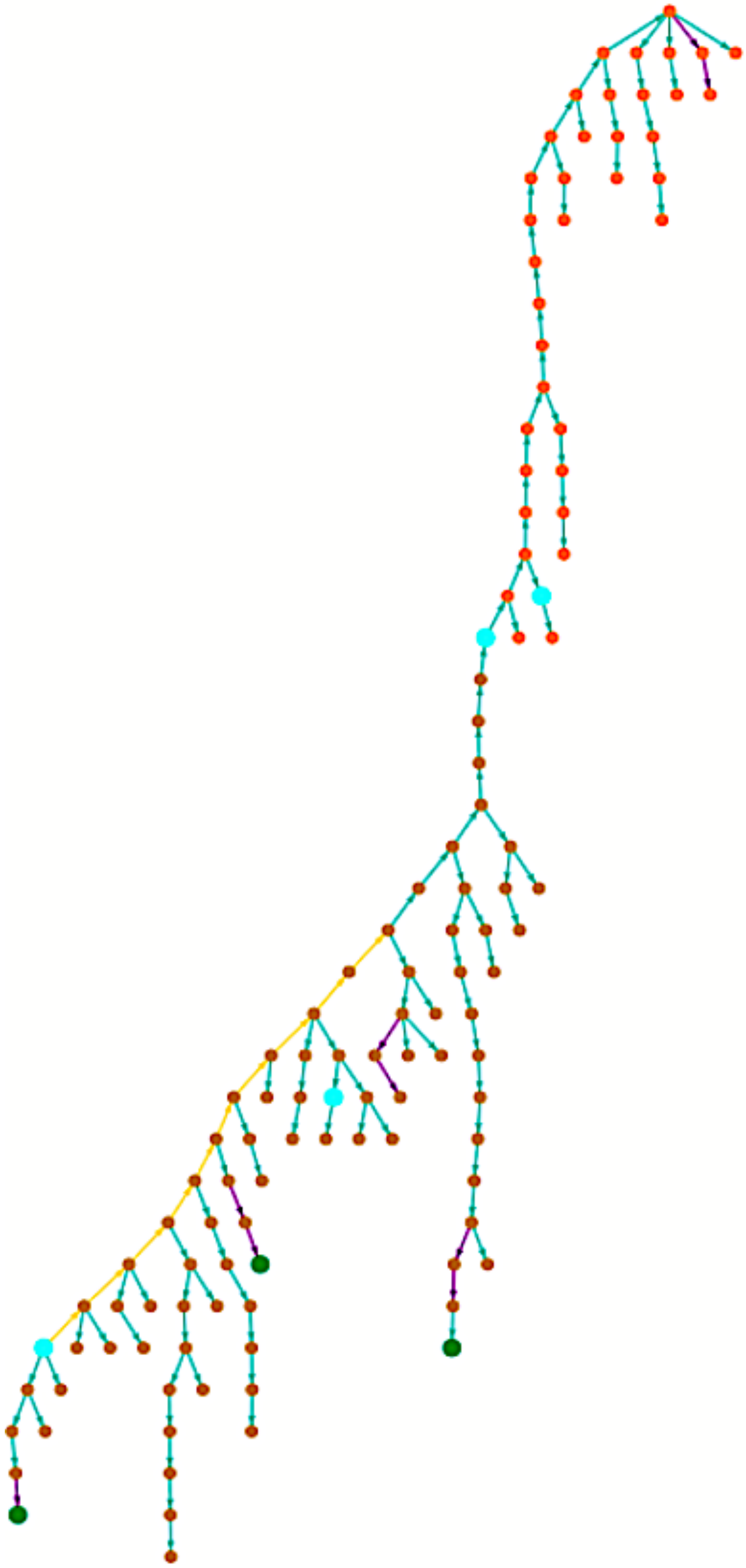


Figure 5.13: Case 9 simulation's tree pattern★
Battery at bus B5 as feeder.

★See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.4 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 10 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.14. See Table 5.30 for the commentary on it.

Table 5.30: Commentary on Case 10 simulation's tree pattern in Figure 5.14[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	None of the four alternative feeders are feeding the grid. The green node at Figure 5.14's top is backup feeder B88. The three green nodes at Figure 5.14's bottom are: The left one is main feeder B1. The middle one is backup feeder B36. The right one is backup feeder B62.
A battery or an EV	cyan	Four nodes are cyan, representing four identical local storages: Three are charging, as one is discharging, feeding the grid during this feeder outage. The node furthest upstream feeds the grid, cyan bus B70, located on the main branch, three lines apart from cyan bus B107. The cyan buses further downstream are in chronological order: B115, and B5. Only buses B5 and B70 are on the main branch.

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	A string of three lines on the main branch, connected to the discharging battery at cyan bus B70.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	None of the alternative feeders are receiving power. A line doesn't transmit to main feeder B1, a line doesn't transmit to bus B35 (connected to backup feeder B36), a string of two lines doesn't transmit to bus B123, a string of three lines doesn't transmit to backup feeder B62, and a string of two lines doesn't transmit to backup feeder B88.

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	cyan	The cyan node furthest upstream is the grid's local storage depleting under this feeder outage, located at bus B70, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	None in this category.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.4 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

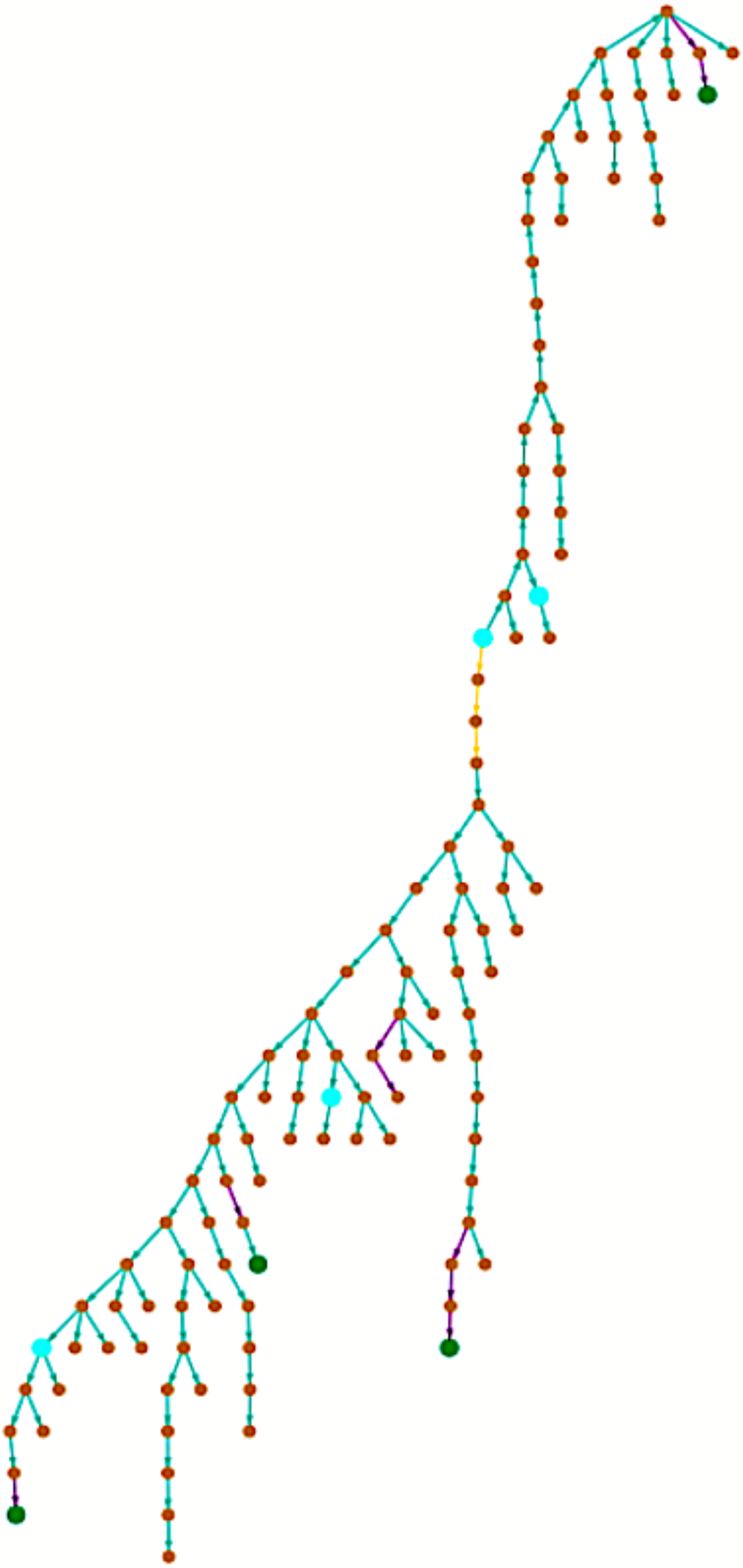


Figure 5.14: Case 10 simulation's tree pattern★
Battery at bus B70 as feeder.

★See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.4 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 11 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.15. See Table 5.31 for the commentary on it.

Table 5.31: Commentary on Case 11 simulation's tree pattern in Figure 5.15★

Significant buses	Color	Categorized buses
An alternative feeder	green	None of the four alternative feeders are feeding the grid. The green node at Figure 5.14's top is backup feeder B88. The three green nodes at Figure 5.14's bottom are: The left one is main feeder B1. The middle one is backup feeder B36. The right one is backup feeder B62.
A battery or an EV	cyan	Four nodes are cyan, representing four identical local storages: Three are charging, as one is discharging, feeding the grid during this feeder outage. The node furthest upstream feeds the grid, cyan bus B107, three lines apart from cyan bus B70 located on the main branch. The cyan buses further downstream are in chronological order: B115, and B5. Only buses B5 and B70 are on the main branch.
Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	The line connected to cyan bus B107, depleting its local storage during this feeder outage.
$40\% < F \leq 60\%$	yellow	A string of five lines on the main branch, one line apart from the cyan bus B107.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	None of the alternative feeders are receiving power. A line doesn't transmit to main feeder B1, a string of two lines doesn't transmit to backup feeder B36, a string of two lines doesn't transmit to bus B123, a string of three lines doesn't transmit to backup feeder B62, and a string of two lines doesn't transmit to backup feeder B88.
Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V <$ 1.1 pu	pink	None in this category.
$V = 1.0$ pu	cyan	The cyan node furthest upstream is the grid's local storage depleting under this feeder outage, located at bus B107, by default set with a voltage of 1.0 pu
0.96 pu $< V <$ 1.0 pu	brown	The rest of the buses.
0.94 pu $< V \leq$ 0.96 pu	orange	None in this category.
0.0 pu $< V \leq$ 0.94 pu	yellow	None in this category.
Zero node potential	violet	None in this category.

★ F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.4 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

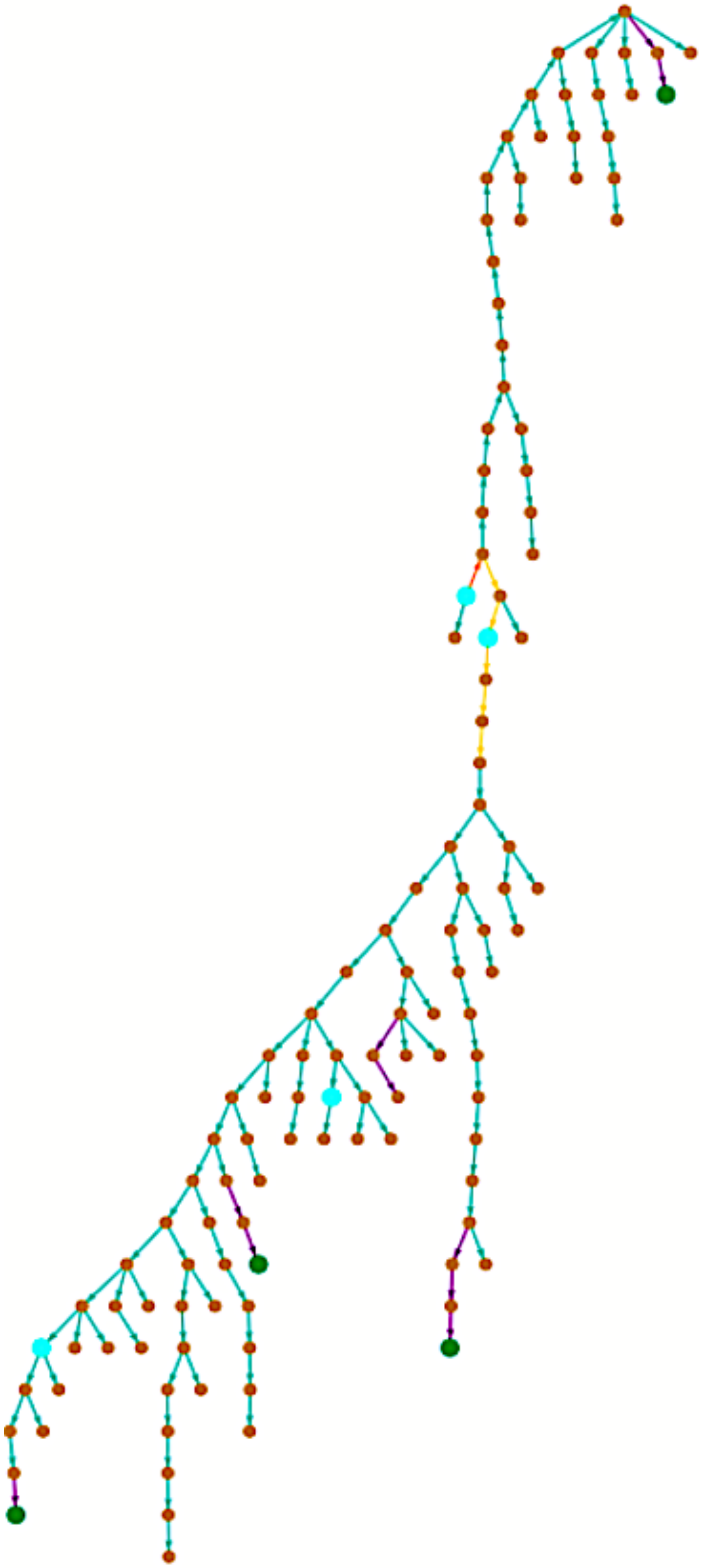


Figure 5.15: Case 11 simulation's tree pattern★
Battery at bus B107 as feeder.

★See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.4 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

Case 12 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.16. See Table 5.32 for the commentary on it.

Table 5.32: Commentary on Case 12 simulation's tree pattern in Figure 5.16[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	None of the four alternative feeders are feeding the grid. The green node at Figure 5.14's top is backup feeder B88. The three green nodes at Figure 5.14's bottom are: The left one is main feeder B1. The middle one is backup feeder B36. The right one is backup feeder B62.
A battery or an EV	cyan	Four nodes are cyan, representing four identical local storages: Three are charging, as one is discharging, feeding the grid during this feeder outage. The node furthest upstream feeds the grid, cyan bus B115, connected to a string of two orange lines. The cyan bus four lines apart from main feeder B1, is bus B5. The other two cyan buses further downstream are in chronological order: B70 and B107. Only buses B115 and B70 are on this grid's main branch.

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	A string of two lines connected to cyan bus B115, depleting its local storage during this feeder outage.
$40\% < F \leq 60\%$	yellow	A string of two lines on the main branch, two lines apart from the cyan bus B115.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	None of the alternative feeders are receiving power. A line doesn't transmit to main feeder B1, a string of two lines doesn't transmit to backup feeder B36, a line doesn't transmit to bus B122, a string of three lines doesn't transmit to backup feeder B62, and a string of two lines doesn't transmit to backup feeder B88.

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	cyan	The cyan node furthest upstream is the grid's local storage depleting under this feeder outage, located at bus B115, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	None in this category.
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.4 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

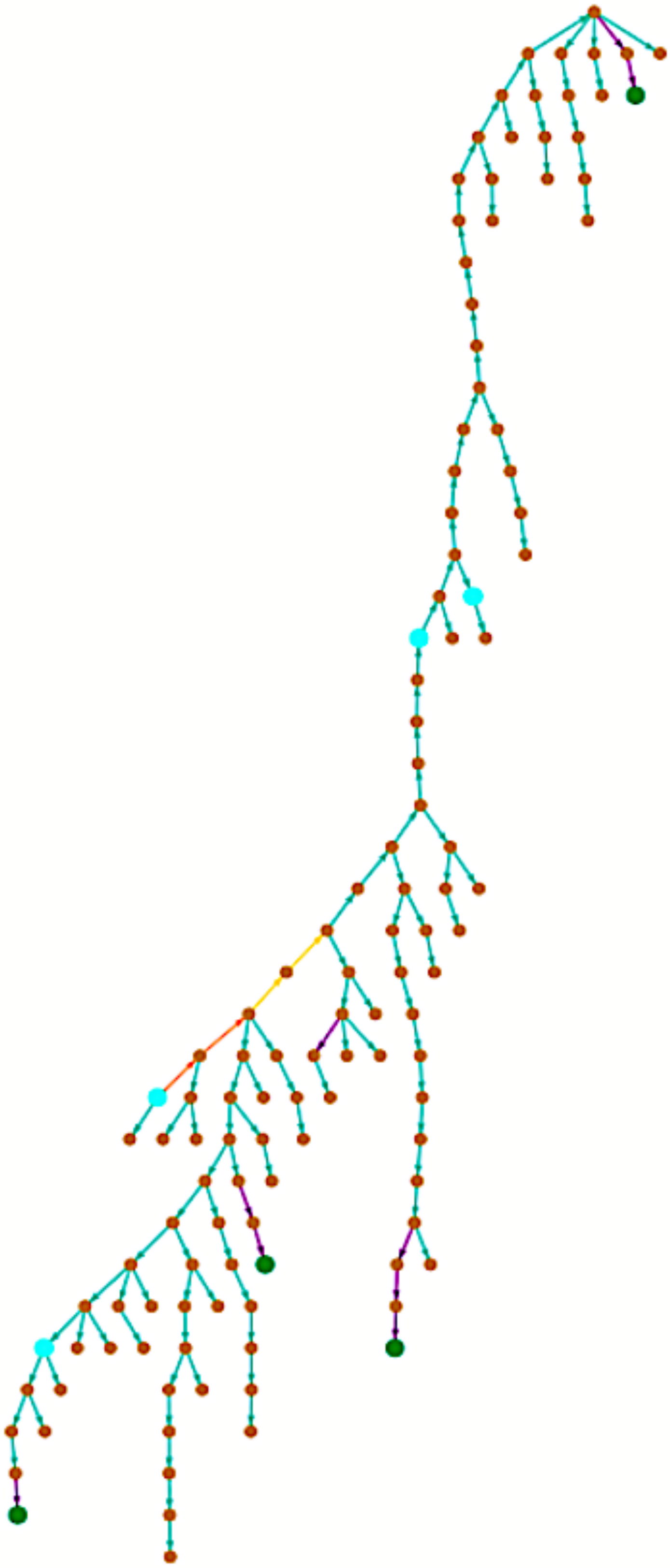


Figure 5.16: Case 12 simulation's tree pattern[★]
Battery at bus B115 as feeder.

[★] See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.4 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

5.4 ... of a battery powered ferry

This simulation scenario has six cases, as seen in Table 3.1. Thus six simulations were performed, as seen in Table 5.35, producing an identical tree pattern. Thus only one tree pattern is seen in this section, randomly choosing Case 17. This scenario's line flows and bus voltages are omitted, downsizing the report.

Case 17 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.17. See Table 5.33 for the commentary on it.

Table 5.33: Commentary on Case 17 simulation's tree pattern in Figure 5.17[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the grid, which is main feeder B1. The smaller green node is backup feeder B36. The other two alternative feeders are orange, connected to violet lines.
A battery or an EV	cyan	The only cyan bus B124, with a charging local storage. The ferry is off grid.

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	None in this category.
$40\% < F \leq 60\%$	yellow	As in Case 1 in Figure 5.2, a string of twelve lines on the main branch, nearly furthest upstream, only a string of two lines between the feeder and the beforementioned string.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A line doesn't transmit to backup feeder B36, a line doesn't transmit to bus B123, a string of three lines doesn't transmit to backup feeder B62, and a string of two lines doesn't transmit to backup feeder B88.

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
1.0 pu $\leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the grid's feeder, main feeder B1, by default set with a voltage of 1.0 pu
0.96 pu $< V < 1.0$ pu	brown	The rest of the buses.
0.94 pu $< V \leq 0.96$ pu	orange	As in Case 1 in Figure 5.2, approximately half of the grid's 124 nodes are colored orange, all connected furthest downstream of the grid (coloring the backup feeders B62 and B88 orange, both connected to violet lines).
0.0 pu $< V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.5 for this simulation scenario's single-line diagram. See Table 5.36 for the indexed simulation commentaries.

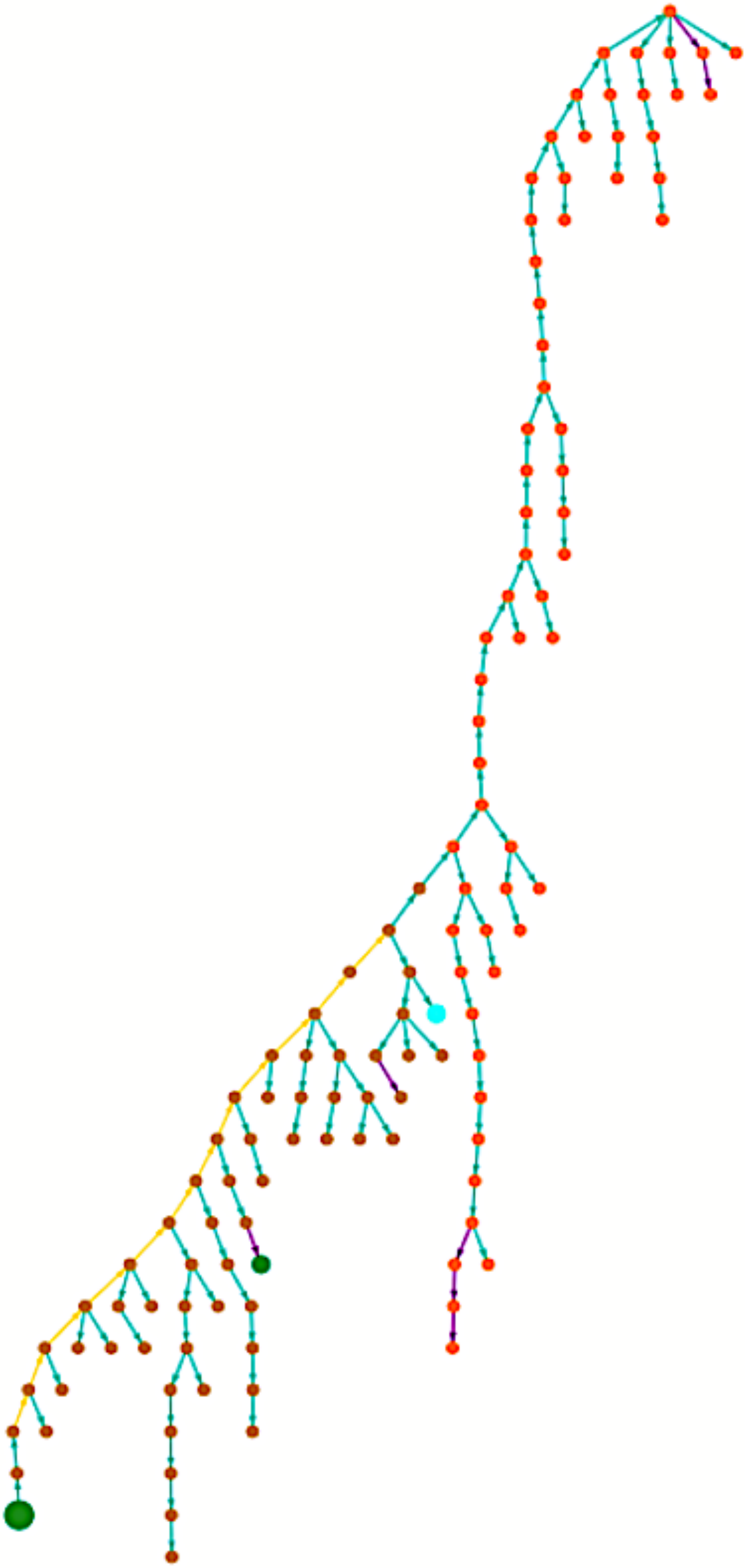


Figure 5.17: Case 17 simulation's tree pattern★
Main feeder B1 as feeder. Onshore battery at bus B124.

★See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.5 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

5.5 ... of vehicles to grid

This simulation scenario has one case, as seen in Table 3.1. Thus one simulation was performed, as seen in Table 5.35. This scenario's line flows and bus voltages are omitted, downsizing the report.

Case 19 simulation's tree pattern

This simulation's tree pattern is seen in Figure 5.18. See Table 5.34 for the commentary on it.

Table 5.34: Commentary on Case 19 simulation's tree pattern in Figure 5.18[★]

Significant buses	Color	Categorized buses
An alternative feeder	green	The largest green node furthest upstream feeds the grid, which is main feeder B1. The smaller green node is backup feeder B36. The other two alternative feeders are orange, connected to violet lines.
A battery or an EV	cyan	Three nodes are cyan, representing three identical plugged in EVs, charging. The cyan node directly downstream of main feeder B1, is bus B2. The cyan node near the grid's middle is bus B48. The last cyan node is bus B117, four lines apart from backup feeder B62.

Line flow [%]	Color	Categorized lines
$F > 100\%$	red	None in this category, thus no overloaded lines.
$80\% < F \leq 100\%$	pink	None in this category.
$60\% < F \leq 80\%$	orange	A string of two lines on the main branch.
$40\% < F \leq 60\%$	yellow	Two strings on the main branch, separated by the orange string. The first string has four lines, connected to main feeder B1. The second string has eight lines.
$0\% < F \leq 40\%$	seagreen	The rest of the lines.
Zero line transmission	violet	A line doesn't transmit to bus B35 (connected to backup feeder B36), a string of two lines doesn't transmit to bus B123, a string of three lines doesn't transmit to backup feeder B62, and a string of two lines doesn't transmit to backup feeder B88.

Bus voltage [pu]	Color	Categorized buses
$V \geq 1.1$ pu	red	None in this category, thus no overloaded nodes.
$1.0 \text{ pu} \leq V < 1.1$ pu	pink	None in this category.
$V = 1.0$ pu	green	The largest green node furthest upstream is the grid's feeder, main feeder B1, by default set with a voltage of 1.0 pu
$0.96 \text{ pu} < V < 1.0$ pu	brown	The rest of the buses.
$0.94 \text{ pu} < V \leq 0.96$ pu	orange	As in Case 1 in Figure 5.2, approximately half of the grid's 124 nodes are colored orange, all connected furthest downstream of the grid (coloring the backup feeders B62 and B88 orange, both connected to violet lines).
$0.0 \text{ pu} < V \leq 0.94$ pu	yellow	None in this category.
Zero node potential	violet	None in this category.

[★] F is a percentage of a line flow divided by its line's capacity. V is a bus voltage. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Table 5.36 for the indexed simulation commentaries.

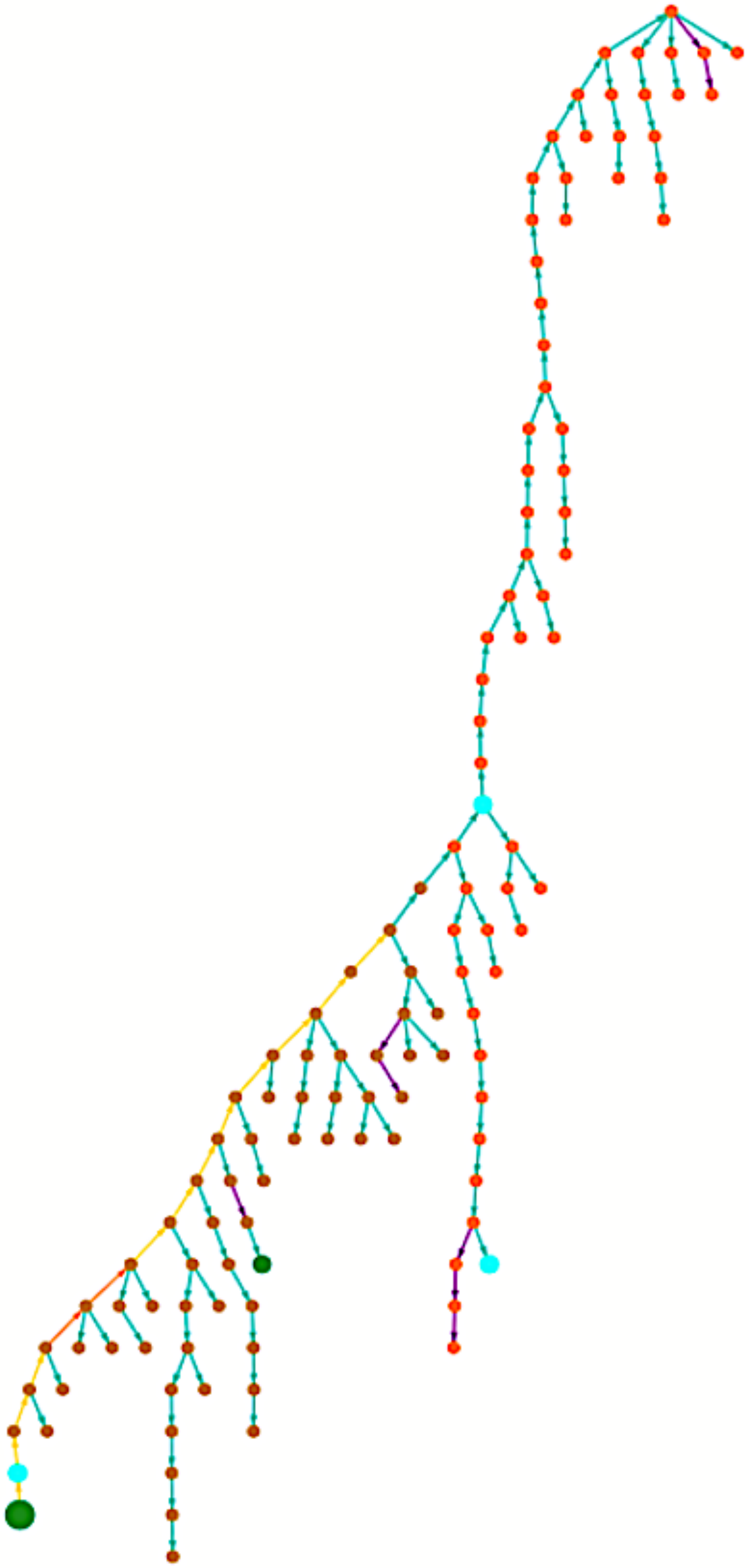


Figure 5.18: Case 19 simulation's tree pattern[★]
Main feeder B1 as feeder. Three identical EVs dispersed at buses B2, B48 and B117.

[★] See Appendix A to enable zooming of this tree pattern. Lines and nodes downstream of violet lines should also be violet, since no power is transmitted to them. See Table 3.1 for an overview of the cases. See Figure 3.6 for this simulation scenario's single-line diagram. See Table 2.3 for color-category explanation. See Table 5.35 for the indexed simulation results.

5.6 Simulation results summary

This thesis analyses a grid (single-line diagram in Figure 1.3), experiencing five scenarios as listed in Table 3.1. In total nineteen versions of the grid are configured. Their simulation results displayed in this report are indexed in Table 5.35. Their commentaries are indexed in Table 5.36.

See Table 3.1 for an overview of the cases.
See an overview of the simulation labels in Table 5.37.

Table 5.35: Indexed simulation results

Case	Line flows	Bus voltages	Tree pattern	Load and loss
Case 1	Tables 5.2, 5.3, 5.39	Tables 5.5, 5.40	Figure 5.2	Table 5.38
Case 2	Tables 5.8, 5.9, 5.39	Tables 5.11, 5.40	Figure 5.4	Table 5.38
Case 3	Tables 5.14, 5.15, 5.39	Tables 5.17, 5.40	Figure 5.6	Table 5.38
Case 4	Tables 5.20, 5.21, 5.39	Tables 5.23, 5.40	Figure 5.8	Table 5.38
Case 5	Table 5.39	Table 5.40	Figure 5.9	Table 5.38
Case 6	Table 5.39	Table 5.40	Figure 5.10	Table 5.38
Case 7	Table 5.39	Table 5.40	Figure 5.11	Table 5.38
Case 8	Table 5.39	Table 5.40	Figure 5.12	Table 5.38
Case 9	Table 5.39	Table 5.40	Figure 5.13	Table 5.38
Case 10	Table 5.39	Table 5.40	Figure 5.14	Table 5.38
Case 11	Table 5.39	Table 5.40	Figure 5.15	Table 5.38
Case 12	Table 5.39	Table 5.40	Figure 5.16	Table 5.38
Case 13	Table 5.39	Table 5.40	Figure 5.17	Table 5.38
Case 14	Table 5.39	Table 5.40	Figure 5.17	Table 5.38
Case 15	Table 5.39	Table 5.40	Figure 5.17	Table 5.38
Case 16	Table 5.39	Table 5.40	Figure 5.17	Table 5.38
Case 17	Table 5.39	Table 5.40	Figure 5.17	Table 5.38
Case 18	Table 5.39	Table 5.40	Figure 5.17	Table 5.38
Case 19	Table 5.39	Table 5.40	Figure 5.18	Table 5.38

Table 5.36: Indexed simulation commentaries

Case	Line flows	Bus voltages	Tree pattern	Load and loss
Case 1	Table 5.1, Section 5.7.2	Table 5.4, Section 5.7.3	Table 5.6	Section 5.7.1
Case 2	Table 5.7, Section 5.7.2	Table 5.10, Section 5.7.3	Table 5.12	Section 5.7.1
Case 3	Table 5.13, Section 5.7.2	Table 5.16, Section 5.7.3	Table 5.18	Section 5.7.1
Case 4	Table 5.19, Section 5.7.2	Table 5.22, Section 5.7.3	Table 5.24	Section 5.7.1
Case 5	Section 5.7.2	Section 5.7.3	Table 5.25	Section 5.7.1
Case 6	Section 5.7.2	Section 5.7.3	Table 5.26	Section 5.7.1
Case 7	Section 5.7.2	Section 5.7.3	Table 5.27	Section 5.7.1
Case 8	Section 5.7.2	Section 5.7.3	Table 5.28	Section 5.7.1
Case 9	Section 5.7.2	Section 5.7.3	Table 5.29	Section 5.7.1
Case 10	Section 5.7.2	Section 5.7.3	Table 5.30	Section 5.7.1
Case 11	Section 5.7.2	Section 5.7.3	Table 5.31	Section 5.7.1
Case 12	Section 5.7.2	Section 5.7.3	Table 5.32	Section 5.7.1
Case 13	Section 5.7.2	Section 5.7.3	Table 5.33	Section 5.7.1
Case 14	Section 5.7.2	Section 5.7.3	-	Section 5.7.1
Case 15	Section 5.7.2	Section 5.7.3	Table 5.33	Section 5.7.1
Case 16	Section 5.7.2	Section 5.7.3	-	Section 5.7.1
Case 17	Section 5.7.2	Section 5.7.3	Table 5.33	Section 5.7.1
Case 18	Section 5.7.2	Section 5.7.3	-	Section 5.7.1
Case 19	Section 5.7.2	Section 5.7.3	Table 5.34	Section 5.7.1

5.7 The shell's summary tables

This section comments on Algorithm B.2's end product (green circle "Display summary tables" in the flow chart in Figure 2.2): the Tables 5.38, 5.39 and 5.40. Respectively, these tables display an overview of every simulation's:

- grid load and loss.
- color-categorized line flows.
- color-categorized bus voltages.

A simulation imitates the impact a case has on the system. A system's impact is quantified by a grid's total consumption and power loss, as well as color-categorized as explained in Table 2.3. In effect, Table 5.38 displays every simulation's grid load- and loss-impact, as Tables 5.39 and 5.40 color-categorize every simulation's line flows and bus voltages respectively.

These three summary tables are implemented in Algorithm B.2 to append a simulation's label in their first column. Thus the labels are explained in Table 5.37, seen on the next page. Also, these summary tables make it easy to compare every case's impact on the use of the system, as well as detect whether any of the cases are either off the charts or stand out.

Table 5.37: The nineteen simulations' labels explained[★]

Case	Simulation label	Scenario	Feeder, topology and any added loads
Case 1	sufeed1	Change of supply bus	Main feeder B1 feeds the grid.
Case 2	sufeed36	Change of supply bus	Backup feeder B36 feeds the grid.
Case 3	sufeed62	Change of supply bus	Backup feeder B62 feeds the grid.
Case 4	sufeed88	Change of supply bus	Backup feeder B88 feeds the grid.
Case 5	spf46t47feed1	Splitting of the grid	The line from bus B46 to B47 is disconnected. Main feeder B1 feeds the left side subgrid.
Case 6	spf46t47feed36	Splitting of the grid	The line from bus B46 to B47 is disconnected. Backup feeder B36 feeds the left side subgrid.
Case 7	spf46t47feed62	Splitting of the grid	The line from bus B46 to B47 is disconnected. Backup feeder B62 feeds the right side subgrid.
Case 8	spf46t47feed88	Splitting of the grid	The line from bus B46 to B47 is disconnected. Backup feeder B88 feeds the right side subgrid.
Case 9	prfeed5	Local storage as backup feeders	Bus B5's battery feeds the grid, as the three other batteries charge.
Case 10	prfeed70	Local storage as backup feeders	Bus B70's battery feeds the grid, as the three other batteries charge.
Case 11	prfeed107	Local storage as backup feeders	Bus B107's battery feeds the grid, as the three other batteries charge.
Case 12	prfeed115	Local storage as backup feeders	Bus B115's battery feeds the grid, as the three other batteries charge.
Case 13	feSfeed1	Battery powered ferry	Main feeder B1 feeds the grid, as bus B124's small battery charges. The ferry is off grid.
Case 14	feSfeed1docks	Battery powered ferry	Main feeder B1 feeds the grid, as bus B124's small battery feeds the charging ferry.
Case 15	feMfeed1	Battery powered ferry	Main feeder B1 feeds the grid, as bus B124's medium battery charges. The ferry is off grid.
Case 16	feMfeed1docks	Battery powered ferry	Main feeder B1 feeds the grid, as bus B124's medium battery feeds the charging ferry.
Case 17	feLfeed1	Battery powered ferry	Main feeder B1 feeds the grid, as bus B124's large battery charges. The ferry is off grid.
Case 18	feLfeed1docks	Battery powered ferry	Main feeder B1 feeds the grid, as bus B124's large battery feeds the charging ferry.
Case 19	v2gfeed1	Vehicles to grid	Main feeder B1 feeds the grid, as three EVs charge.

[★] The title column of Tables 5.38, 5.39 and 5.40 consists of these simulation labels.
See Table 3.1 for an overview of the cases.
See Table 5.35 for the indexed simulation results.
See Table 5.36 for the indexed simulation commentaries.

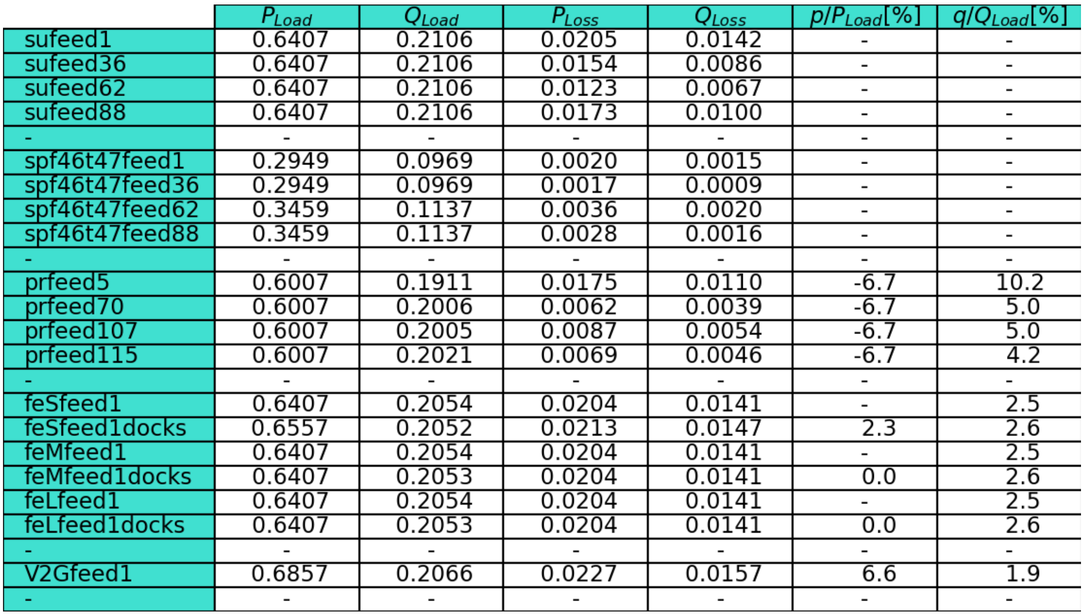
5.7.1 Grid load and loss and any added loads

Commenting on Table 5.38, firstly its title column consists of every simulation's labels, implemented in Algorithm B.2 and detailed in Table 5.37. Table 5.38 displays every simulation's grid load and loss, and the percentage of any added loads. It is one of the three summary tables Algorithm B.2 produces as its end product (green circle "Display summary tables" in the flow chart of Figure 2.2). Thus a user receives an overview of every case's grid load and loss statistics, quantifying every case's stamp.

The two first scenarios were not implemented to incorporate added loads, thus their grid load stays fixed and their percentages of added loads are zero. Their change of feeder though resulted in a changed grid loss, as expected with the changed line flow direction. The three latter scenarios all include specifically voltage dependent loads. A value appears in the percentage column of added active power when the case concerned was implemented to either decrease (supply) or increase (consumption) its active load in the shell's either feeder- or injection-loop (Algorithm B.2's flow chart in Figure 2.2). A value appears in the percentage column of added reactive power contribution when the case concerned was implemented to calculate either a decrease or an increase of its reactive load (Section 1.6.3).

Table 5.38 shows that the last (Case 19, Figure 3.6) simulation has the largest grid both load and loss. This was the scenario vehicles to grid's only case, which of all this thesis' cases had the largest amount of active power consumption implemented, as stated also in the table's column for percentage of active load increase (6.6%). The active power consumption of the charging local storages or onshore battery in the other cases were overlooked in this thesis. Comparing the simulations having added loads, this last simulation has the least amount of added reactive power contribution (1.9%), fitting since an EV has a smaller capacity than a battery, thus has less impact on the system. Although there are several (three) EVs plugged into the grid, overall they strain the grid less than a battery does. The parameters of Table 1.4 and the changes made in this thesis to two equations concerning Equation 2 appear thus to be valid.

Table 5.38: Every simulation's total power load and loss, and the percentage of added loads★



	P_{Load}	Q_{Load}	P_{Loss}	Q_{Loss}	p/P_{Load}[%]	q/Q_{Load}[%]
sufeed1	0.6407	0.2106	0.0205	0.0142	-	-
sufeed36	0.6407	0.2106	0.0154	0.0086	-	-
sufeed62	0.6407	0.2106	0.0123	0.0067	-	-
sufeed88	0.6407	0.2106	0.0173	0.0100	-	-
-	-	-	-	-	-	-
spf46t47feed1	0.2949	0.0969	0.0020	0.0015	-	-
spf46t47feed36	0.2949	0.0969	0.0017	0.0009	-	-
spf46t47feed62	0.3459	0.1137	0.0036	0.0020	-	-
spf46t47feed88	0.3459	0.1137	0.0028	0.0016	-	-
-	-	-	-	-	-	-
prfeed5	0.6007	0.1911	0.0175	0.0110	-6.7	10.2
prfeed70	0.6007	0.2006	0.0062	0.0039	-6.7	5.0
prfeed107	0.6007	0.2005	0.0087	0.0054	-6.7	5.0
prfeed115	0.6007	0.2021	0.0069	0.0046	-6.7	4.2
-	-	-	-	-	-	-
feSfeed1	0.6407	0.2054	0.0204	0.0141	-	2.5
feSfeed1docks	0.6557	0.2052	0.0213	0.0147	2.3	2.6
feMfeed1	0.6407	0.2054	0.0204	0.0141	-	2.5
feMfeed1docks	0.6407	0.2053	0.0204	0.0141	0.0	2.6
feLfeed1	0.6407	0.2054	0.0204	0.0141	-	2.5
feLfeed1docks	0.6407	0.2053	0.0204	0.0141	0.0	2.6
-	-	-	-	-	-	-
V2Gfeed1	0.6857	0.2066	0.0227	0.0157	6.6	1.9
-	-	-	-	-	-	-

Otherwise the load- and loss-impact do not differ much from simulation to simulation, except for the simulations with a split grid, having approximately half the standard load. Adding the grid load of the left subnetwork (Figure 3.2a) to the right subnetwork (Figure 3.2b) equals the first

---

★See Table 5.37 for simulation label explanation. Return to Section 5.7.

scenario's grid load. The sixth (Case 6, Figure 3.2a) simulation has the smallest grid loss. The left side subnetwork fed by backup feeder B36, and right side subnetwork fed by backup feeder B88, is the system's split-mode pairing with the smallest grid loss of 0.0045 pu active and 0.0025 pu reactive.

This thesis' added active power contributions are a load decrease for the scenario of local storage as backup feeders (Figure 3.4) and a load increase for the two latter scenarios (Figure 3.5 and Figure 3.6). All the cases of the scenario of local storage as backup feeder have the same amount of load decrease (-6.7%), as this scenario was implemented to alternate its four local storages as backup feeders, thus feeding the grid from different vantage points than its original alternative feeders (Figure 3.1). The scenario of the battery powered ferry (Figure 3.5) has one case where the onshore battery is too small to feed the charging ferry, thus the grid experiences a net load increase of 2.3% more active power consumption. The two latter instances of the ferry's demand being met is represented with a net 0.0%. Considering this, the latter instance was not valid in illustrating an onshore battery meeting the ferry's demand in abundance. A larger battery should contribute more reactive power than a medium and small sized one. Thus it appears that as the onshore battery's size increased its charging slope should have increased accordingly.

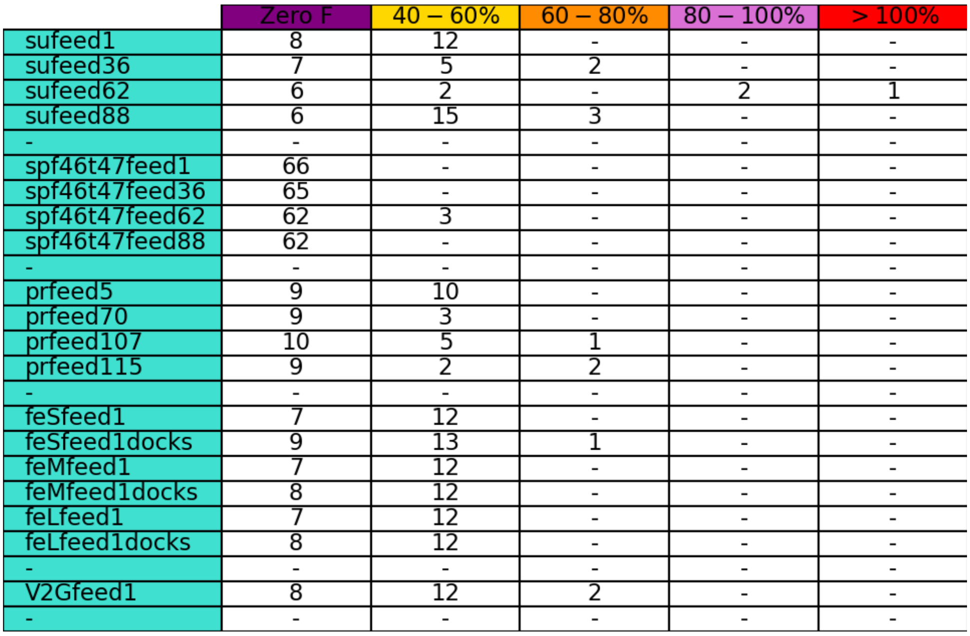
This thesis' added reactive power contributions are all positive. Interpreting this, the local storages, onshore battery and EVs all transmit directionless power into the grid in these snapshots of the electrified grid. The largest one (Case 9, Figure 3.4) is 10% of its reactive grid load, meaning this case introduces a tenth more electrical noise to the grid. This alerts the user to consider employing noise-reducing measures.

5.7.2 Color-categorized line flows

Commenting on Table 5.39, firstly its title column consists of every simulation's labels, implemented in Algorithm B.2 and detailed in Table 5.37. Table 5.39 displays every simulation's color-categorized line flows, excluding the seagreen category of "a flow of 0-40% of a line's flow capacity" in Table 2.3. It is one of the three summary tables Algorithm B.2 produces as its end product (green circle "Display summary tables" in the flow chart in Figure 2.2). Thus a user receives an overview of every case's statistics on line flows, similar to a user comparing every tree pattern's lines. A tree pattern displays their locations, while this table offers a rough overview of the system's line flow profile.

In total the grid contains 123 lines. A row in Table 5.39 shows the amount of lines never transmitting and those transmitting more than 40% of their line's capacity. It becomes clear that there are more lines without than within categorization, and the flow-impact does not differ much from case to case. The first and three latter scenario's simulations all have close to ten lines not transmitting (Figure 3.1). Only the scenario of splitting the grid (Figure 3.2a and Figure 3.2b) have a higher amount of lines never transmitting. This has to do with the implementation of the scenario (Section 4.2), in effect leaving the other subgrid barren while analysing the current subgrid. Only the third simulation has a line overflowing its capacity (Figure 5.6).

Table 5.39: Color-categorized line flows of every simulation★



	Zero F	40 – 60%	60 – 80%	80 – 100%	> 100%
sufeed1	8	12	-	-	-
sufeed36	7	5	2	-	-
sufeed62	6	2	-	2	1
sufeed88	6	15	3	-	-
-	-	-	-	-	-
spf46t47feed1	66	-	-	-	-
spf46t47feed36	65	-	-	-	-
spf46t47feed62	62	3	-	-	-
spf46t47feed88	62	-	-	-	-
-	-	-	-	-	-
prfeed5	9	10	-	-	-
prfeed70	9	3	-	-	-
prfeed107	10	5	1	-	-
prfeed115	9	2	2	-	-
-	-	-	-	-	-
feSfeed1	7	12	-	-	-
feSfeed1docks	9	13	1	-	-
feMfeed1	7	12	-	-	-
feMfeed1docks	8	12	-	-	-
feLfeed1	7	12	-	-	-
feLfeed1docks	8	12	-	-	-
-	-	-	-	-	-
V2Gfeed1	8	12	2	-	-
-	-	-	-	-	-

5.7.3 Color-categorized node voltages

Commenting on Table 5.40, firstly its title column consists of every simulation's labels, implemented in Algorithm B.2 and detailed in Table 5.37. Table 5.40 displays every simulation's color-categorized node voltages, excluding the brown category of "a bus voltage greater than 0.96 pu and smaller than 1.0 pu" in Table 2.3. It is one of the three summary tables Algorithm B.2 produces as its end product (green circle "Display summary tables" in the flow chart of Figure 2.2). Thus a user receives an overview of every case's statistics on bus voltages, similar to a user comparing every tree pattern's nodes. A tree pattern displays their locations, while this table offers a rough overview of the system's voltage profile.

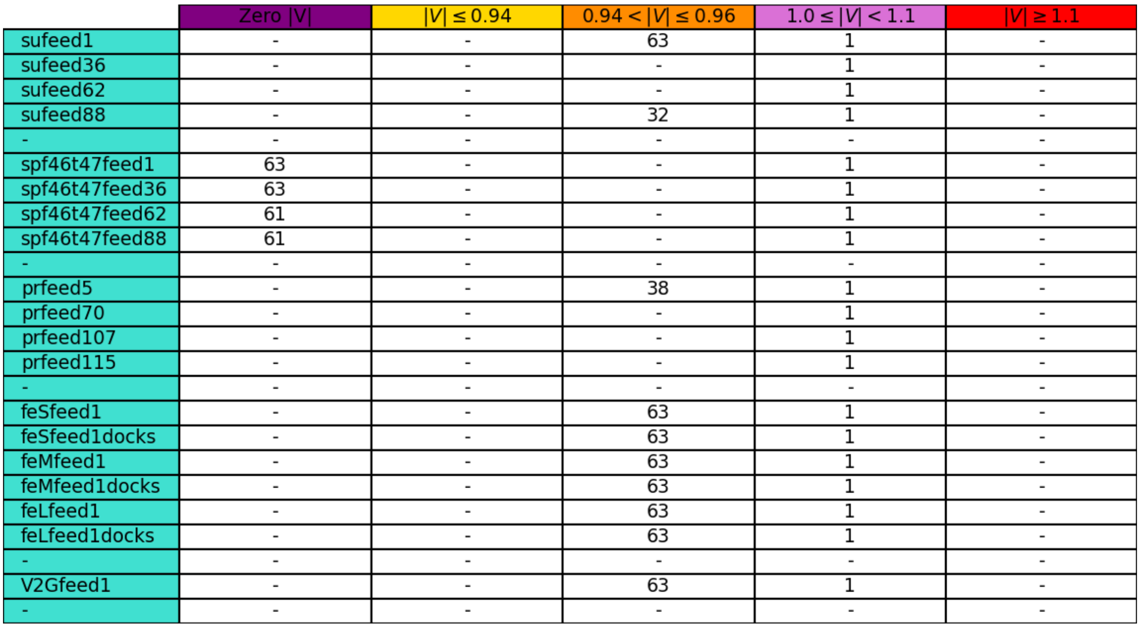
★See Table 5.37 for simulation label explanation. Return to Section 5.7.

In total the grid contains 124 nodes (buses). A row in Table 5.40 shows the amount of nodes in an outage or with a categorized potential. It becomes clear that all grids, except for the left-side subgrid (Figure 3.2a), supplied by main feeder B1, have 63 buses in orange category (voltage magnitude larger than 0.94 pu and smaller than or equal to 0.96 pu). Two other cases have orange buses: The grid supplied by backup feeder B88 and the grid supplied by a battery at bus B5. Bus B88 is at the opposite end of the grid of main feeder B1, while bus B5 is four lines apart from main feeder B1. Both have approximately one third of its buses colored orange. Thus all the cases with orange buses have a similar main branch, only the case with backup feeder B88 has the opposite line flow direction.

All cases have one bus in pink category (voltage magnitude larger than or equal to 1.0 pu and smaller than 1.1 pu). The implementation of bypassing this category if the bus is supplying the grid (has a voltage magnitude of exactly 1.0 pu and a voltage angle of exactly 0.0 pu) does not affect this summary table, because Algorithm B.3's function *tableplot* is implemented to bypass the pink category when a table's title column starts with a "B", as a voltage table does in this report. As discovered in the scenario of a change of supply's (Figure 3.1) Cases 2 and 3 in their respective Tables 5.11 and 5.17, displaying their bus voltages, both have an additional voltage within pink category. The reason why this other pink voltage didn't get categorized as pink has of yet not been found.

Only the split grid cases have buses with zero voltage, belonging to the subgrid left barren when simulating the other subgrid. Not a single bus has either too low (less than or equal to 0.94 pu, yellow) or too high voltage (equal to or greater than 1.1 pu, red). The voltage-impact differs mainly in the orange category.

Table 5.40: Color-categorized bus voltages of every simulation[★]



	Zero $|V|$	$|V| \leq 0.94$	$0.94 < |V| \leq 0.96$	$1.0 \leq |V| < 1.1$	$|V| \geq 1.1$
sufeed1	-	-	63	1	-
sufeed36	-	-	-	1	-
sufeed62	-	-	-	1	-
sufeed88	-	-	32	1	-
-	-	-	-	-	-
spf46t47feed1	63	-	-	1	-
spf46t47feed36	63	-	-	1	-
spf46t47feed62	61	-	-	1	-
spf46t47feed88	61	-	-	1	-
-	-	-	-	-	-
prfeed5	-	-	38	1	-
prfeed70	-	-	-	1	-
prfeed107	-	-	-	1	-
prfeed115	-	-	-	1	-
-	-	-	-	-	-
feSfeed1	-	-	63	1	-
feSfeed1docks	-	-	63	1	-
feMfeed1	-	-	63	1	-
feMfeed1docks	-	-	63	1	-
feLfeed1	-	-	63	1	-
feLfeed1docks	-	-	63	1	-
-	-	-	-	-	-
V2Gfeed1	-	-	63	1	-
-	-	-	-	-	-

[★] See Table 5.37 for simulation label explanation. Return to Section 5.7.

6 Future software development

6.1 Future shell development

A flat start of the system should be optional. Thus the network configuration should exclude the function *flatStart*, having nothing to do with configuring the topology, explained in Section 6.2. Its exclusion is illustrated in Figure 6.1, where a yellow ellipse is an in-/output. *flatStart* has been removed (from the feeder-loop into the injection-loop), and put into the flow chart in Figure 6.2.

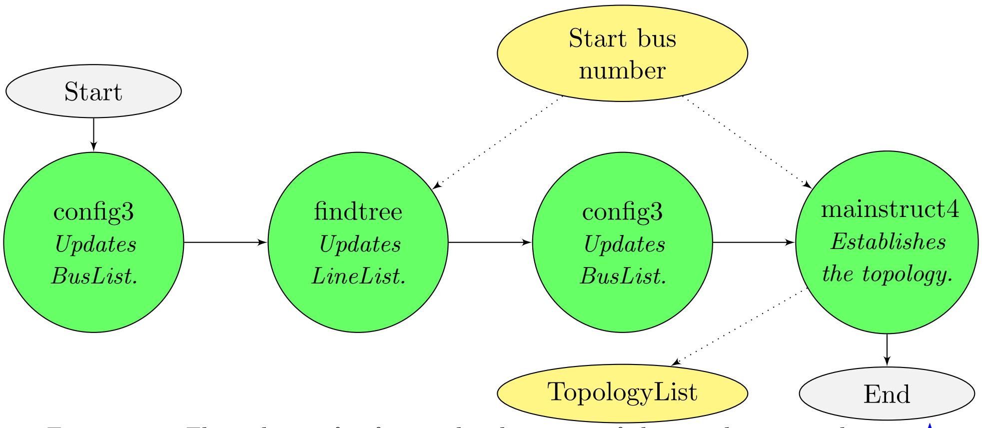


Figure 6.1: Flow chart of a future development of the topology initialization★

The tool supports multiple simulations on the same network revolving different load profiles. Thus, a rerun of the injection-loop either with or without a flat start of the grid, should be optional, as illustrated in the flow chart in Figure 6.2; a future development from the flow chart in Figure 2.2. The orange circles, the "rerun (flat)"-arrow and the single yellow ellipse state the future development of the shell. This flow chart omits all other in-/outputs, downsizing it.

As of now, *flatStart* was discovered at the end of writing this report to have been written within the function *DistLF* (Section 1.2). In other words *flatStart* is actually executed twice in Algorithm B.2. Illustrating it with the flow chart in Figure 2.2, the first execution takes place in the green circle "Initialize topology", and the last execution takes place in the green circle "Run simulation". This is similar to the flow chart in Figure 6.2, without the optional rerun bypassing a flat start of the system.

Also, the optional bus power change should only be applied within the injection-loop. Thus the topology and the simulation is further distinguished than before, setting them apart as two entities. At the moment the bus injection update is done within both the feeder- and injection-loop.

6.1.1 ... of a change of supply

Regarding this scenario, no further development comes to mind.

6.1.2 ... of splitting of the grid

In principle, the grid could be split in three subgrids or more, i.e. with just one subgrid connected to an interconnected grid. This thesis has only focused on illustrating how PyDSAL splits a grid, but it is possible to implement a split anywhere in the grid, and calculate the resulting load flow.

Downsizing this report, only one split network was analysed, but there is already implemented in Algorithm B.2 (specifically line 91, containing a long list of names of several networks, and lines

---

★See Figure 6.2 for a flow chart of a future development of Algorithm B.2, overwriting this flow chart with its orange circle "Configure topology".

127-139, making use of all of these names in stating which line is to be disconnected) a demo for analysing several split networks, but all of them split only one line. Splitting more than one of a grid's lines, could be implemented by making a list of the lines to be disconnected. Visiting every line in LineList (Section 2.1), as these disconnected lines are detected they are declared to be in an outage, zeroing the line's parameter *ibstat* (Section 4.2).

If a subgrid doesn't have any feeder, this part of the grid will experience a blackout. Implementing an activation of a local storage as a backup feeder, enables the software with its failure in finding a feeder, to detect a substitute in the grid, switching it on to supply-mode. Thus increasing the flexibility of power system operations, enabling PyDSAL to self-heal the blackout. This requires the scenarios splitting of the grid and local storages as backup feeders to be meshed.

6.1.3 ... of local storages as backup feeders

The three other local storages charging should have had a active power consumption implemented, but this was overlooked in this thesis.

A battery is a depleting supply, thus an implementation of the grid's evolvement over time would increase PyDSAL's flexibility, monitoring the power system operation. This could be implemented as several snapshots of the grid experiencing a stage of the battery's depletion. Thus the user could watch a tree pattern morph from one state to another. An implementation of this could be to list all the snapshots, and then execute a slide show of them.

The class object *battery* (Algorithm B.4) contains battery attributes of integers. In order to represent a battery's different stages, these attributes should be lists. As a battery discharges many of its attributes will change. Thus the first state would be the first element in all the lists, the second state would be the second element in all the lists etc. As a consequence, the injection-loop (Section 2.1) reiterates until all depletion stages are fulfilled, since these stages are injection updates.

Additionally, a battery's capacity in both MW and pu should be presented to the user, added to a tree pattern (Section 6.2). Thus easier for the user to form an idea of this battery's impact on the grid. The attribute $Estorage$ (Algorithm B.4) is probably intended for such use, but has not yet been utilized in PyDSAL.

Implementing this, a new command could be set at Algorithm B.2's Fork (gray ellipse in the flow charts in Figures 2.2 and 6.2): exit if the list of stages is completed, otherwise rerun with or without a flat start of the system. This list of stages is illustrated as a yellow ellipse in the flow chart in Figure 6.2.

A discharging battery consequently loses its potential as it gives away what it had stored. Thus power electronics ensure that the voltage magnitude is kept at the same level throughout the discharging. Due to the difficulties this entails, the battery's potential should be higher than PyDSAL's default setting of 1.0 pu. This requires further development of Algorithm B.3's function $flatStart$ (Section 6.2).

6.1.4 ... of a battery powered ferry

The ferry plugging in its onboard battery to the onshore battery, should be implemented as an extra battery connecting to the bus. In effect, the bus's attribute *battery* (Section 4.4) should as a default be implemented as an empty list rather than a zeroed integer, enabling several batteries to connect to a bus. This requires further development of Algorithm B.3's function *getload* (Section 6.2).

6.1.5 ... of vehicles to grid

Implementing an EV charging station or allowing several V2Gs to connect to a bus, requires a bus's attribute $v2g$ (Section 4.5) as a default to be an empty list rather than a zeroed integer. This requires further development of Algorithm B.3's function *getload* (Section 6.2).

Concerning the results of the scenario vehichles only case (Case 19), every EV's voltage was smaller than its reference voltage, since reactive power was stored rather than produced (Section 1.6.3): ΔQ^{ctrl} was positive (the column to the far right in Table 5.38). Thus the voltage reference should have been lower than 1.0 pu, probably as low as 0.95 pu. A table should have been made of the V2G voltages, to compare them with their respective bus voltages.

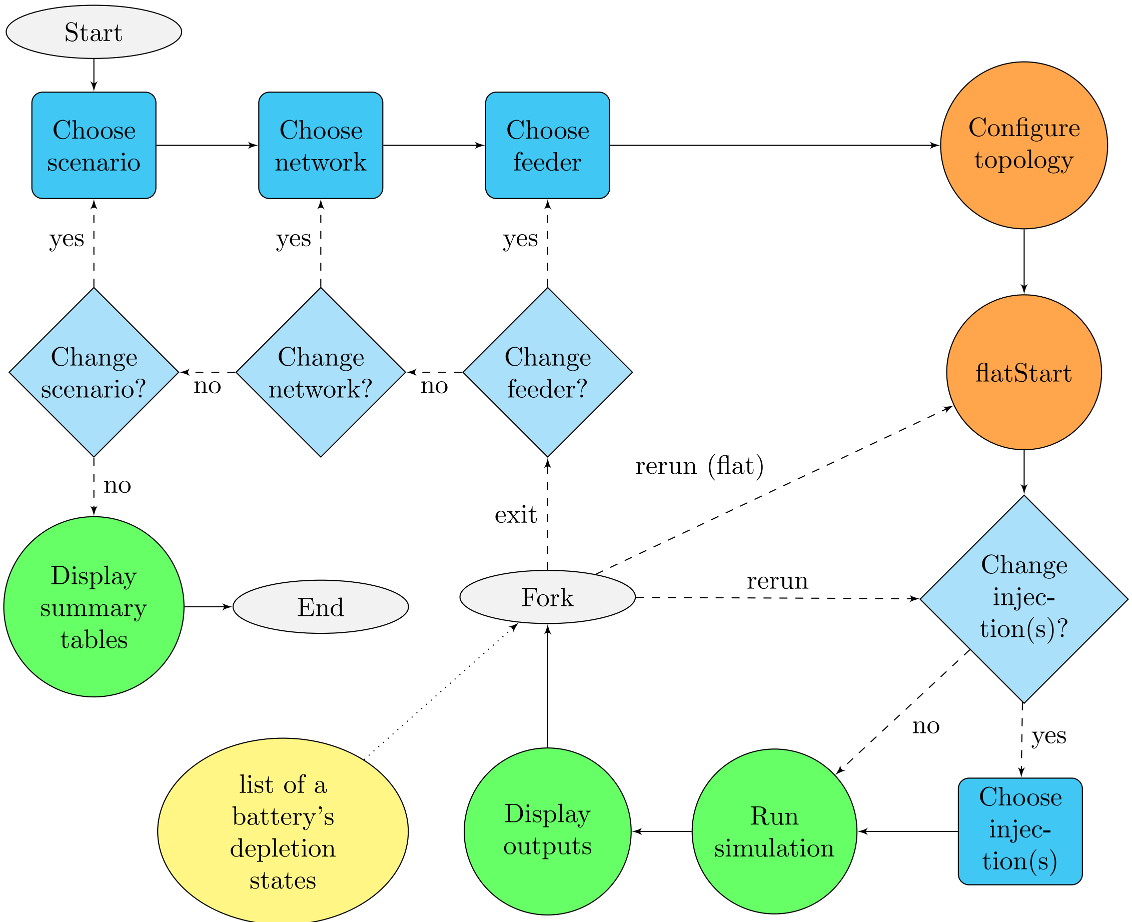


Figure 6.2: Flow chart of a future development of Algorithm B.2[★]

[★] See Figure 2.2 for Algorithm B.2's flow chart.
See Figure 6.1 for a flow chart of a future development of the topology initialization, overwritten by the orange circle "Configure topology" in this flow chart.

6.2 Future function development

6.2.1 *flatStart*

To fulfill the demands set in Section 6.1, the function *flatStart* must be updated. As its name entails, it doesn't incorporate any objects into neither BusList nor LineList. Meaning, it has nothing to do with the topology. When the user requires a network with no record of power flowing in its lines, *flatStart* resets (flattens) every bus's electric parameters of:

- Voltage magnitude and angle
- Accumulated load and loss

These are the parameters altered during a simulation by FBS (Section 1.3), except for the sensitivities. Thus *flatStart* should also reset the sensitivities. Otherwise within the injection-loop the last simulation's sensitivities overlap the next simulation, affecting PyDSAL's calculation of any added voltage dependent loads in the grid (Section 1.6.3).

Additionally, *flatStart*'s resetting of voltages should be updated when a battery is acting as a backup feeder, supplying the system. The bus this battery is connected to, states the system's start bus. In effect, the start bus's voltage should have a higher voltage than the default value of 1.0 pu. Thus taking into consideration a battery's struggle to maintain its voltage magnitude, delivering its charge to the grid. Probably a magnitude of 1.05 pu would suffice, singling out this depleting supply.

6.2.2 *getload*

To fulfill the demands set in Section 6.1.4 and Section 6.1.5 the function *getload* must be implemented to process a list rather than just an integer as it does at the moment. It is called by the function *accload* (Section 1.3) and executed in Algorithm B.2's injection-loop within the green circle "Run simulation" in the flow charts in Figures 2.2 and 6.2.

Thus a node should be able to include several objects such as batteries and EVs. Requested by *accload* to visit a node, *getload* fishes for a list of objects concerning this node during its visit. If a list is caught, every object's power contribution to this node is estimated. Having processed all the listed objects, the function adds them to the load already stored in the node (stored following the importation of BusList in Algorithm B.2's network-loop, flow chart in Figure 2.2, Section 2.1). The latter part was already implemented as this thesis began, as well as having one EV and one battery both connected to the same bus.

6.2.3 *dispTree*

Commenting on the tree patterns displayed in this report, several issues became apparent:

- The visibility of alternative feeders, local storages, the ferry and EVs.
- The visibility of sizes of loads. Should be able to with a glance locate small, medium and large loads.
- The coloring of specific buses clashed with the coloring of lines/nodes (Table 2.3), making the color-code counter-intuitive.

The bus numbers of the grid's alternative feeders should be visible even when the user has zoomed out to a bird's eye view of the tree pattern. In other words, the tree pattern graphics in this thesis should have shown the alternative feeders' bus numbers in the same text size as the rest of the report. Also, every node's net injection should be made clear to the user, letting the user spot the grid's "fountains" and "sinkholes" with a glance.

Thus these node categories could be introduced:

- A circular node for a node with a net negative injection.
- A square node for a node having the capacity to achieve a net negative injection.
- A triangular node for the remaining nodes.

Could be confusing to have too many shapes to contend with, thus only three shapes seem fitting. With such an update of a tree pattern, the color-category for specific buses may be removed. Implementing this, *dispTree* must be extended to include processing of a node's injection value prior to drawing the node.

Thus the supply node would be circular rather than larger than the other nodes. Considering a future version of PyDSAL that includes the discharging of EVs etc., analysing the impact of e.g. households selling power to the grid: if any of these respective nodes result in a net negative injection, they will be easy to spot when they are circular. The location of the circular nodes implies whether it is an alternative feeder or a substitute. A substitute could be a depleting battery, or a pool of depleting EVs connected to one bus. It would be of interest to implement say 50 EVs connected to the grid, dispersed, all discharging into the grid. Probably few of the affected nodes turn circular, but their nodes will definitely be square shaped. Usually, the substitutes are dispersed, while the alternative feeders are at the grid's periphery.

6.2.4 ... of the color-categorization

Commenting on the simulation results displayed in this report (Table 5.35), several issues became apparent:

- Lines and nodes downstream of violet (not transmitting power) lines, weren't violet (in an outage).
- The coloring of voltage and line flow categories were counter-intuitive, since a bus's voltage must be kept within a range for system stability, while a line must avoid overflowing.

One solution could be to have fewer categories and/or never use the same color twice.

The violet coloring of lines and nodes was implemented by marking the lines transmitting a flow of exactly 0.0 pu to be violet. As 1 MW is 0.0000001 pu in this thesis (Section 1.5.2), the violet criteria should be for flows greater than -0.0000001 pu and smaller than 0.0000001 pu.

7 Conclusion

This master thesis further developed the object-oriented software PyDSAL (Python Distribution System Analysis Library) and was used to study several grid configurations, based on the test system CINELDI 124.

The five scenarios investigated were as follows:

1. A change of supply bus.
2. Splitting of the grid, with a change of supply bus.
3. Local storages as backup feeders one by one.
4. The battery powered ferry's intermittent loading of the grid, with change of onshore battery.
5. Vehicles to grid, charging.

Thus the grid CINELDI 124 underwent several transformations. The two first scenarios had straightforward procedures to accomplish. The three latter scenarios introduced complexities that proved challenging to overcome, i.e. requiring the user to set the electrical and topological parameters for the local storages and voltage dependent loads to be incorporated into the grid.

Investigations of nineteen grid cases implemented in the Python language, resulted in a new type of shell for PyDSAL, overwriting its previous shell. In effect, the shell was systematized with loops within loops, stating the tool's four main sequences. Thus a user may tune in to this strategic guide. This expanded user-friendliness enables a flexible approach to studying alternative topologies and supply situations in a distribution grid or a microgrid.

A prototype for simulating a radial grid's single-line diagram displaying attributes was further developed. Via a HTML file, this zoomable tree pattern depicts flow directions, labels and color-categorized load flow characteristics.

Bibliography

[Haq95] M. H. Haque. ‘Load flow solution of distribution systems with voltage dependent load models’. In: *Electric Power Systems Research 36 (1996) 151-156* (1995).

[Cop19] Oxford University Press Copyright © 2010. *Oxford Dictionary of English*. Copyright © 2005–2019 Apple Inc. Apple macOS Catalina 10.15’s Dictionary app version 2.3.0 (239.5), 2019.

[Fos20] Olav Bjarte Fosso. ‘PyDSAL - Python Distribution System Analysis Library’. In: *DOI: 10.1109/POWERCON48463.2020.9230554* (2020).

[AS] Nord Pool AS. *System price*. URL: https://www.nordpoolgroup.com/Market-data1/Dayahead/Area-Prices/SYS1/Hourly/?view=chart (visited on 04/01/2022).

[FME] CINELDI FME. *Centre for Intelligent Electricity Distribution*. URL: https://www.sintef.no/projectweb/cineldi/ (visited on 03/02/2022).

A HTML files of tree patterns

Every tree pattern was saved in a HTML file, enabling zooming of the grid when opened in a web browser. These ninenteen HTML files are delivered together with this PDF file of this report.

See Table 5.35 for the indexed simulation results. See Section 2.2.1 for more details on the tool's drawing of a tree pattern.

B PyDSAL

A PyDSAL's previous version of the function *dispTree*

See Table 1.2 for an overview of PyDSAL's scripts. Algorithm B.1 is commented on in Section 2.2.1.

Algorithm B.1: The function dispGraph

```
def dispGraph(self, topologyList,top=1, feeders=[], LEC=[], charging=[],
↪ lowVolt=[], overload= [],disconnected=[]):
    """
    Builds and display the graph as a HTML-file
    """

    self.BuildGraph(topologyList,top=1,feeders=feeders, LEC=LEC,
    ↪ charging=charging, lowVolt=lowVolt )
    self.AddEdges(topologyList, overload=overload,disconnected=disconnected)
    nt.from_nx(nx_graph)
    nt.show("nt.html")



# Visit all nodes in the forward list.
def BuildGraph(self, topologyList,top=1, feeders=[], LEC=[], charging=[],
↪ lowVolt=[]):
    """Visit all nodes in a forward approach and build the graphic
    ↪ representation
    """
    #  from pyvis.network import Network
    #  nt = Network('1000px', '2000px', layout=None)
    def adaptNode(node, top=1, feeders=[], LEC=[], charging=[],lowVolt=[]):
        if node == top:
            nx_graph.add_node(node, label="Main feeder", color='green')
        elif node in feeders:
            nx_graph.add_node(node,label= "Bck feeder", color='green')
        elif node in LEC:
            nx_graph.add_node(node,label= "LEC", color='#FF33F9')
        elif node in charging:
            nx_graph.add_node(node,label= "Charging", color='purple')
        elif node in lowVolt:
            nx_graph.add_node(node,label= "Low Volt", color='yellow')
        else:
            nx_graph.add_node(node)
#     print(feeders, LEC, lowVolt)
    for x in topologyList:
        if len(x) > 1:
            # print('Bus' + str(x[0].busnum))
        #    nt.add_node(int(x[0].busnum))
            adaptNode(int(x[0].busnum), top=1, feeders=feeders, LEC=LEC,
            ↪ charging=charging, lowVolt=lowVolt)
            iloop = 1
            while iloop < len(x):  # Do for all branches of a bus
                self.BuildGraph(x[iloop], top=1, feeders=feeders, LEC=LEC,
                ↪ charging=charging, lowVolt=lowVolt)
                iloop += 1
        else:
```

```
            #   print('Bus' + str(x[0].busnum))
          #  nt.add_node(int(x[0].busnum))
             adaptNode(int(x[0].busnum), top=1, feeders=feeders, LEC=LEC,
              ↪  charging=charging, lowVolt=lowVolt)



# Visit all nodes in the reverse list.
def AddEdges(self, topologyList, overload=[], disconnected=[]):
    """ Visit all the nodes in a backward approach and prints the Bus name
    """
    def adaptEdge(node1, node2,  overload=[], disconnected=[]):
        if (node1, node2) in overload:
            nx_graph.add_edge(node1, node2, color='red', value=2)
        elif (node1,node2) in disconnected:
            nx_graph.add_edge(node1, node2, color='brown', value=2)
        else:
            nx_graph.add_edge(node1, node2, color=" #33AFFF", arrow=True)

    for x in reversed(topologyList):
        if len(x) > 1:
          #  print('Bus' + str(x[0].busnum))
            if x[0].toline:
     #           nt.add_edge(int(x[0].toline.fbus),int(x[0].busnum))
                adaptEdge(int(x[0].toline.fbus), int(x[0].busnum),
                ↪  overload=overload, disconnected=disconnected)

            iloop = 1
            while iloop < len(x):  # Do for all branches of a bus
                self.AddEdges(x[iloop], overload=overload,
                ↪  disconnected=disconnected)

                iloop += 1
        else:
     #       print('Bus' + str(x[0].busnum))
            if x[0].toline:
            #    nt.add_edge(int(x[0].toline.fbus), int(x[0].busnum))
                adaptEdge(int(x[0].toline.fbus), int(x[0].busnum),
                ↪  overload=overload, disconnected=disconnected)
```

B PyDSAL's shell

See Table 1.2 for an overview of PyDSAL's scripts. Algorithm B.2 is commented on in Section 2.1. It was further developed from the previous version of PyDSAL's shell, located here. A flow chart illustrates this algorithm in Figure 2.2.

Algorithm B.2: `concept.py`

```
# Import functions from .py file:
from DistLoadFlow_vIngrid import *



# Set parameters:
dpbattery = -0.04 # Set the battery's injection for its feeding the grid:
dpferry = .03 # Twice of dpV2G # Set the Ferry's load:
dpV2G = .015# Set the V2G's load:

pbatmax = .04
qbatmax = .04
sizebatonshore = [0.5, 1.0, 1.5]
pferrymax = np.multiply(dpferry, sizebatonshore)
qferrymax = np.multiply(dpferry, sizebatonshore)
pV2Gmax = .04
qV2Gmax = .04

slopebat = .2 # Discharges what the grid drains from it.
slopeV2G = .05 # Charges for hours.

batterynr = [5, 70, 107, 115]
ferrynr = 124
V2Gnr = [2, 48, 117]

# Set the battery on the shore's injection (discharge):
dpbatteryonshore = np.multiply(-1,pferrymax)
dpbatL = -1*dpferry
dpbatteryonshore[-1] = dpbatL
print('Battery menu of injections on shore:',dpbatteryonshore)

# Create BatteryList, BatterySizes and V2GList :
# cmode = 1 is used in UpdateVolt, when BusList[itr].iloss==1.
# Meaning that, loss minimization script is activiated at these specific buses,
↪  and only with a battery having cmode=1 will get a loss minimization.
# cmode = 2 is used in potential() and getload().
BatteryList = [Battery(
    bus = nr,
    cmode = 2,# I guess cmode stands for controlmode.
    svcstat = 1.0,
    vref = 1.0,
    injPmax = pbatmax,
    injPmin = 0.0,
    injQmax = qbatmax,
    injQmin = 0.0,
    slopeP = slopebat,
    slopeQ = slopebat)
    for nr in batterynr]
```

```
BatterySizes = [Battery(
    bus = ferrynr,
    cmode = 2,# I guess cmode stands for controlmode.
    svcstat = 1.0,
    vref = 1.0,
    injPmax = pferrymax[i],
    injPmin = 0.0,
    injQmax = qferrymax[i],
    injQmin = 0.0,
    slopeP = slopebat,
    slopeQ = slopebat)
    for i in range(len(pferrymax))]

V2GList = [V2G(
    bus = nr,
    cmode = 2,
    v2gstat = 1,
    vref = 1,
    injPmax = pV2Gmax,
    injPmin = 0.0,
    injQmax = qV2Gmax,
    injQmin = 0.0,
    slopeP = slopeV2G,
    slopeQ = slopeV2G)
    for nr in V2Gnr]



# Show impact:
impact=[] #To be appended at bottom of this script.
impactcountF=[] #To be appended at bottom of this script.
impactcountV=[] #To be appended at bottom of this script.
rowno=[] #To be appended at bottom of this script.

col = ['$P_{Load}$', '$Q_{Load}$', '$P_{Loss}$','$Q_{Loss}$']
addloads = ['$p/P_{Load}$[%]', '$q/Q_{Load}$[%]']
col += addloads

colcountF = ['Zero F', '$40-60\%$', '$60-80\%$', '$80-100\%$', '$>100\%$']
colcountV = ['Zero |V|', '$|V|\leq 0.94$', '$0.94<|V|\leq 0.96$', '$1.0\leq
↪  |V|<1.1$', '$|V|\geq 1.1$']


# Choose a scenario:
#scenarios = ['supply']#, 'split', 'provision', 'ferry', 'V2G']
#for ind in range(1):
scenarios = ['supply', 'split', 'provision', 'ferry', 'V2G']
for ind in range(5):
    Scenario = scenarios[ind]
    print('')
    print('')
    print('')
    print('')
    print('--- Investigating:',Scenario,'---')
    print('')
    print('')
    print('')
```

```
    # Name network:
    names = ['']
    if Scenario=='split':
        #names = ['f42t44','f44t45', 'f45t46', 'f46t47']
        names = ['f46t47']
    if Scenario=='ferry':
        names = ['S', 'M', 'L']
    if Scenario=='V2G':
        names = ['G'] # Because label, as defined near bottom of this code,
        ↪   starts with VG for the V2G investigation. Add 'G' here, and the label
        ↪   starts with 'V2G'.

    k=0
    # Choose network:
    for name in names:
        print('')
        print('')
        print('')
        print('--- Running network',name+': ---')

        # Import data from Excel file; in effect calibrating BusList and
        ↪   LineList:
        BusList, LineList = BuildSystem3() # Cannot be put outside of this
        ↪   name-loop, because otherwise values from last network case overlap
        ↪   the current one.

        # Update BusList:
        if Scenario=='provision':
            i=0
            for batobj in BatteryList:
                busnr = batobj.bus
                BusList[busnr-1].battery = BatteryList[i]
                i+=1
        if Scenario=='ferry':
            indx = names.index(name)
            BusList[ferrynr-1].battery = BatterySizes[indx]
        if Scenario=='V2G':
            i=0
            for V2Gobj in V2GList:
                busnr = V2Gobj.bus
                BusList[busnr-1].v2g = V2GList[i] #V2Gs dock.
                i+=1

        # Update LineList:
        num=1
        for lobj in LineList:
            lobj.linenum = num #Their linenumbers are used in dispTree.
            num += 1
```

```
        if Scenario=='split':
            if name[-1]=='4':
                splitfbus = 42
                splittbus = 44
            if name[-1]=='5':
                splitfbus = 44
                splittbus = 45
            if name[-1]=='6':
                splitfbus = 45
                splittbus = 46
            if name[-1]=='7':
                splitfbus = 46
                splittbus = 47
            for lobj in LineList:
                if lobj.fbus==splitfbus and lobj.tbus==splittbus:
                    lobj.ibstat = 0
                    print('')
                    print('***** Disconnected line between buses', lobj.fbus,'and', lobj.tbus, '*****')
                    print('')



        # Create network with latest update of BusList and LineList:
        N = DistLoadFlow3(BusList, LineList) # Every network N has two lists each: One BusList and one LineList.



        # Choose feeder:
            #In the case of Scenario=='split':
            # Make N become the subsystem to the left by choosing a startBus left of sep.point:
            # Make N become the subsystem to the right by choosing a startBus right of sep.point:
        if Scenario=='provision':
            feeders = [BusList[i-1] for i in batterynr]
        elif Scenario=='ferry' or Scenario=='V2G':
            feeders = [BusList[0]]
        else:
            nr = [1, 36, 62, 88]
            feeders = [BusList[i-1] for i in nr]

        for feeder in feeders:
            # Initialize:
            N.flatStart()
            N.config3()
            N.findtree(feeder.busnum)
            N.config3()
            N.topology = N.mainstruct4(startBus = feeder.busnum)

            print('')
            print('')
            print('Supplying the load from Bus' + str(feeder.busnum) + ':')
            print('Length of mainlist:',len(N.topology))
            print('Last bus:',N.topology[-1][0].busname)
            #N.ForwardSearch(N.topology)
```

```
        # Set display-lists:
        batteries=[]
        ferry=[]
        V2Gs=[]
        charging=[]
        if Scenario=='provision':
            batteries = feeders
            chargingnr=[]
            for i in batterynr:
                if i!=feeder.busnum:
                    chargingnr.append(i)
            charging = [BusList[i-1] for i in chargingnr]
        if Scenario=='V2G':
            V2Gs = [BusList[i-1] for i in V2Gnr]
            charging = V2Gs


        # Choose power:
        if Scenario=='provision':
            print('')
            print('The battery at Bus' + str(feeder.busnum),'discharges:')
            N.changePower(feeder.busnum, dpbattery)
        if Scenario=='V2G':
            print('')
            print('V2Gs dock:')
            for V2Gobj in V2GList:
                N.changePower(V2Gobj.bus, dpV2G) # One V2G docks at each
                 ↪  respective bus.


        # Choose how many rounds of load flows:
        docks='' #Used in label below.
        rounds = ['1st']
        if Scenario=='ferry':
            rounds = rounds + ['2nd']


        # Run distribution load flow:
        for run in rounds: #Only the ferry scenario has len(rounds)=2.
            if Scenario=='ferry':
                charging = [BusList[ferrynr-1]]
                # The N-object, which is now being analyzed, includes only
                 ↪  one battery, which is meant for feeding the ferry,
                 ↪  therefore it is charging when the ferry is off grid.
                if run=='2nd':
                    docks = 'docks'
                    print('')
                    print('The ferry docs:')
                    charging=[]
                    ferry = [BusList[ferrynr-1]] # Now that the ferry
                     ↪  connects to the grid, it appears in the resulting
                     ↪  .html-file.
```

```
                        dp = dpferry + dpbatteryonshore[indx]
                        N.changePower(ferrynr, dp)




                # Case description is completed. Activate load flow simulation:
                PQList = N.DistLF(epsilon=0.00001) #Oppdatere med updategen etter
                ↪  DistLF? Fant ikke den funksjonen i Fosso sin
                ↪  DistLoadFlow_v2.py



                N.resetBuses()
                FLists = N.checkFlow() #Marked line flows.
                VLists = N.checkVolt() #Marked bus voltages.




                # Reset power as well as accumulate the added loads:
                pinj=0
                qinj=0
                if Scenario=='provision':
                    pinj = dpbattery
                    for batobj in BatteryList:
                        qinj += batobj.qinj
    #                    batobj.qinj = 0
                    dpreset = np.multiply(-1,dpbattery)
                    N.changePower(feeder.busnum, dpreset)
                if Scenario=='ferry':
                    qinj = BusList[ferrynr-1].battery.qinj
#                    BusList[ferrynr-1].battery.qinj = 0
                    if run=='2nd':
                        pinj = dp
                        dpreset = np.multiply(-1,dp)
                        N.changePower(ferrynr, dpreset)
                        BusList[ferrynr-1].battery = 0 #Two load flow runs
                        ↪  completed, next up is the new battery size, therefore
                        ↪  clearing the current battery.
                if Scenario=='V2G':
                    pinj = dpV2G*len(V2GList)
                    dpreset = np.multiply(-1,dpV2G)
                    for V2Gobj in V2GList:
                        qinj += V2Gobj.qinj
#                        V2Gobj.qinj = 0
                        N.changePower(V2Gobj.bus, dpreset) # One V2G disconnects
                        ↪  at each respective bus.





                # Display results:
                label = Scenario[0:2] + name + 'feed' + str(feeder.busnum) +
                ↪  docks
                print('---',label, 'analysis completed ---')
```

```
                # Create lists for dispTree:
                CINELDI124feeders = [BusList[i-1] for i in [1, 36, 62, 88]]
                tag_bobj = [CINELDI124feeders, batteries, V2Gs, ferry, charging, [feeder]] + VLists

                #Draw radial tree:
                N.dispTree(N.topology, tagBus=tag_bobj, tagLine=FLists, filename=label)

                #Tabulate voltage and/or flow results:
                k=1
                if k==0:
#                    fr=0
#                    N.dispFlow(fromLine=fr, tpres=True, case=label, FLists=FLists)
#                    N.dispVolt(fromBus=fr, tpres=True, case=label, VLists=VLists)
                    for i in range(2):
                        fr = i*62
                        N.dispFlow(fromLine=fr, tpres=True, case=label, FLists=FLists)
                        N.dispVolt(fromBus=fr, tpres=True, case=label, VLists=VLists)
                        #break

                    if Scenario=='supply':
                        input('pause; Press Enter to continue.') # To avoid overheating.
#                k=1

                #Create lists for impact-tables:
                if PQList==[]:
                    PQList = ['?']*len(col) #No solution was found.
                else:
                    padd = (pinj/float(PQList[0]))*100
                    qadd = (qinj/float(PQList[1]))*100
                    if padd==0 and run=='1st':
                        padd='-'
                    else:
                        padd = '{:7.1f}'.format(padd)
                    if qadd==0 and run=='1st':
                        qadd='-'
                    else:
                        qadd = '{:7.1f}'.format(qadd)
                    PQList += [padd, qadd]

                row=[]
                for FList in FLists:
                    temp = '-'
                    l = len(FList)
                    if l > 0:
                        temp = str(l)
                    row.append(temp)
                impactcountF.append(row)

                row=[]
                for VList in VLists:
                    temp = '-'
```

```
                    l = len(VList)
                    if l > 0:
                        temp = str(l)
                    row.append(temp)
                impactcountV.append(row)

                rowno.append(label)
                impact.append(PQList)
                # Exiting load flow loop
            # Exiting feeder loop
        if Scenario!='ferry' or (name=='L' and run=='2nd'):
            rowno.append('-')
            dashes = ['-']*len(col)
            impact.append(dashes)
            dashes = ['-']*len(colcountF)
            impactcountF.append(dashes)
            dashes = ['-']*len(colcountV)
            impactcountV.append(dashes)
        # Exiting names of network loop
    # Exiting investigation loop
#print(col)
#print(impact)
#print(colcountF)
#print(impactcountF)
#print(colcountV)
#print(impactcountV)
N.tableplot(impact, columns=col, rows=rowno, case='impact')
N.tableplot(impactcountF, columns=colcountF, rows=rowno, case='impactcountF')
N.tableplot(impactcountV, columns=colcountV, rows=rowno, case='impactcountV')
```

C PyDSAL’s laws and functions

See Table 1.2 for an overview of PyDSAL’s scripts. Algorithm B.3 is commented on in Section 2.2.

Algorithm B.3: `DistLoadFlow-vIngrid.py`

```
# Copyright (c) 2021, Olav B. Fosso, NTNU
#
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without modification,
# are permitted provided that the following conditions are met:
#
#     * Redistributions of source code must retain the above copyright notice,
#       this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above copyright notice,
#       this list of conditions and the following disclaimer in the documentation
#       and/or other materials provided with the distribution.

import math
import matplotlib.pyplot as plt

import DistribObjects_vIngrid
from BuildSystem_vIngrid import *


# Graphics representation
percentS = .4
percentM = .6
percentL = .8
from pyvis.network import Network
import networkx as nx
charge = 'cyan'
large= 'orchid'
medium = 'darkorange'
over = 'red'
small = 'gold'
lec = '#FF33F9'
zero = 'purple'
supply = 'forestgreen'
buses = 'darkgoldenrod'
lines = 'lightseagreen'
cases = 'turquoise'


class DistLoadFlow3:
    """
    Common base class Radial System's (Distribution)  Load Flow
    Input:
        BusList      - List off all Bus objects
        LineList     - List of all transmission lines objects
    Returns: None
    """
```

```
    def __init__(self, Buses, Lines):
        self.BusList = Buses
        self.LineList = Lines
        self.voang = np.zeros(len(self.BusList))
        self.vomag = np.ones(len(self.BusList))
        self.topology = []

    def config3(self):
        """Function for making the topology - it sets up the connection between
        ↪   two buses by assigned the line to the to bus
        and by preparing a list of from bus connections (branching)
        Problem: Currently turn the direction of too many lines when the
↪   connection point splits the chain
        """
        self.clearTopology()
        for lobj in self.LineList:
            if lobj.ibstat:
                itr = lobj.tbus - 1
                ifr = lobj.fbus - 1
                self.BusList[ifr].tolinelist.append(lobj)
                self.BusList[
                    itr].toline = lobj  # Add information to each bus of a line
                    ↪   abouth which line that connects the meighbour bus.
                self.BusList[ifr].fromline = lobj

        # Add the topology information needed to define the tree structure
        for lobj in self.LineList:
            if lobj.ibstat:
                itr = lobj.tbus - 1
                ifr = lobj.fbus - 1
                self.BusList[ifr].nextbus.append(
                    self.BusList[itr])  # Add the next bus to the list of
                    ↪   branches of the bus

    def findtree(self, bstart=1):
        """ Finds a trestructure from a spesified node
            The from and two nodes are switched to get a positive flow
↪   direction.
        """
        def mswitch(ifrom, ito):  # To switch direction
            return ito, ifrom

        def direct(bindex, val =None):  # Recursive function for topology search
        ↪   and direction of a graph.
            ibus = self.BusList[bindex]
            for lobj in lineconnlist[bindex]:
                if lobj.tbus == ibus.busnum:
                    lobj.fbus, lobj.tbus = mswitch(lobj.fbus, lobj.tbus)
                if lobj not in lineconnlist[lobj.tbus - 1]:
                    print('Grid is not radial')
                    return False
                lineconnlist[lobj.tbus - 1].remove(
                    lobj)  # When a line is checked remove the object from the
                    ↪   lineconnlist of the to-bus
                val = direct(lobj.tbus - 1)
                if val == False:
                    return val
```

```
        lineconnlist = []  # Define and initialize with sublists
        iloop = 0
        while iloop < len(self.BusList):
            lineconnlist.append([])
            iloop += 1

        # Find lines connected to all buses
        for lobj in self.LineList:
            if lobj.ibstat:
                itr = lobj.tbus - 1
                ifr = lobj.fbus - 1
                lineconnlist[ifr].append(lobj)
                lineconnlist[itr].append(lobj)

        # Build a tree structure
        ibus = self.BusList[bstart - 1]  # Identify the bus object to start with
        valid = direct(bstart - 1)
        return valid

    # Flat start
    def flatStart(self):
        iloop = 0
        while iloop < len(self.BusList):
            ibus = self.BusList[iloop]
            ibus.vomag = 1.0
            ibus.voang = 0.0
            ibus.ploadds = 0.0
            ibus.qloadds = 0.0
            ibus.pblossds = 0.0
            ibus.qblossds = 0.0
            iloop += 1

    # Set up a list for the main branch, where subbranches are stored as sublists. Handles all radial topologies
    def mainstruct4(self, startBus=None):
        """
        An algorithm to establish a tree structure based on the system data. Sets up a list for the main branch,
        with sublists wherever branching occurs. The algorithm can handle any radial topology, but not meshed grids.
        """
        if startBus is None:
            startBus = self.BusList[0]
        else:
            startBus = self.BusList[startBus - 1]
        mainlist = []                               # Make the main branch
        nextobj = [startBus]                        # Set next object to the first bus
        while len(nextobj) > 0:                     # Until we reach the end of the main branch
            if len(nextobj) == 1:                   # If no branch is present, add the bus to main branch
                mainlist.append(nextobj)
            if len(nextobj) > 1:
                mainlist.append([nextobj[0]])       # If branches occur, add the root bus to the main branch
                for i in range(1, len(nextobj)):    # Go through each sub branch
                    bra = self.branch4(nextobj, i)  # Make sub branches
```

```
                    mainlist[-2].append(bra)        # Add sub branch to the root bus
                nextobj = mainlist[-1][0].nextbus       # Set next bus to the next in main branch
        return mainlist

    def branch4(self, nextobj, i):
        """
        A recursive algorithm to follow every branch until the end. In case of sub branches, the algorithm calls itself.
        """
        sub = [[nextobj[i]]]                                # Make the sub branch, and add the first bus
        nextobj = sub[-1][0].nextbus                        # Set next bus to the first of the branch
        while len(nextobj) > 0:                             # Follow until the end of the sub branch
            if len(nextobj) == 1:                           # If no further branching, add to sub branch
                sub.append(nextobj)
            if len(nextobj) > 1:
                sub.append([nextobj[0]])                    # If further branching, add root of branch to sub branch
                for j in range(1, len(nextobj)):
                    subsub = self.branch4(nextobj, j)   # Go through each subsub branch(recursive step)
                    sub[-2].append(subsub)              # Add possible subsub branches
            nextobj = sub[-1][0].nextbus                    # Set next bus to next bus in sub branch
        return sub

    # Return the buses connected to the grid
    def connectedBuses(self, topologyList):
        """
        The function returns a list of all buses connected to the grid.
        """
        buses = []
        for x in topologyList:
            if len(x) > 1:
                buses.append(x[0])
                iloop = 1
                while iloop < len(x):  # Do for all branches of a bus
                    am = self.connectedBuses(x[iloop])
                    for i in range(0, len(am)):
                        buses.append(am[i])
                    iloop += 1
            else:
                buses.append(x[0])
        return buses

    # Clear topology to start new configuration of the grid
    def clearTopology(self):
        """
        The function clears all topology parameters to ensure correct configuration when the system is altered.
        """
        for bus in self.BusList:
```

```
                bus.connectedLines = []
                bus.tolinelist = []
                bus.toline = 0
                bus.fromline = 0
                bus.nextbus = []

        # Connect a line
        def connectLine2(self, line):
            """
            Connects a line. Can take a line object or a line index as input.
            """
            lineindex = 0
            if type(line) is Line:
                lineindex = self.LineList.index(line)
            if type(line) is int:
                lineindex = line
            self.LineList[lineindex].ibstat = 1
            print('Connected line between bus ' + str(self.LineList[lineindex].fbus)
            ↪  + ' and ' + str(
                self.LineList[lineindex].tbus))

        # Disconnect a line
        def disconnectLine2(self, line):
            """
            Disconnects a line. Can take a line object or a line index as input.
            """
            lineindex = 0
            if type(line) is Line:
                lineindex = self.LineList.index(line)
            if type(line) is int:
                lineindex = line
            self.LineList[lineindex].ibstat = 0
            print('Disconnected line between bus ' +
            ↪  str(self.LineList[lineindex].fbus) + ' and ' +
                 str(self.LineList[lineindex].tbus))

        #Disconnect a bus
        def disconnectBus(self, busnum):
            """
            The functions disconnects a bus from the system by disconnecting all
↪       lines connected to it, and resetting
            the voltage magnitude and angle.
            """
            bind = busnum - 1
            bus = self.BusList[bind]
            self.disconnectLine2(self.LineList.index(bus.toline))
            self.BusList[bind].toline = 0
            for lobj in bus.tolinelist:
                self.disconnectLine2(self.LineList.index(lobj))
            self.BusList[bind].vomag = 0.0
            self.BusList[bind].voang = 0.0

        # Disconnect all overloaded buses
        def disconnectBuses(self, buses):
            """
            The function goes through a list of buses and disconnects them.
            """
            for bus in buses:
```

```
            self.disconnectBus(bus.busnum)
        print('Disconnected bus : ', [o.busnum for o in buses])

    # Check is any buses have too high or too low voltage
#Ingrid    def checkOverLoad(self):
    def checkVolt(self): #Ingrid
        """
        The function goes through the list of buses to check for over- and underloaded buses.
        Returns: All buses that have been under- and overloaded
        """
        zero=[]
        under=[]
        medium=[]
        large=[]
        over=[]
        for bobj in self.BusList:
#Ingrid            if bus.vmax < bus.vomag:
            v = bobj.vomag
            if v==0.0:
                zero.append(bobj)
            if v <= .94 and v!=0.0:
                under.append(bobj) #Like small
            if v > .94 and v <= .96:
                medium.append(bobj)
            if v >= 1.0 and v < 1.1:
                large.append(bobj)
            if v >= 1.1:
                over.append(bobj)

        if len(over) > 0:
            print('Overload found at bus: ', [o.busnum for o in over])
        else:
            print('No overload found.') #Ingrid
        if len(under) > 0:
            print('Underload found at bus: ', [u.busnum for u in under])
        else:
            print('No underload found.') #Ingrid
        return [zero, under, medium, large, over]

    #Reset buses not in the topology
    def resetBuses(self):
        """
        Sets the voltage magnitude and angle of all buses not connected to the grid to zero for display purposes.
        """
        top = self.connectedBuses(self.topology)
        for bus in self.BusList:
            if bus not in top:
                bus.vomag = 0.0
                bus.voang = 0.0

    #Change power consumption at a bus
    def changePower(self, busnum, delta):
        """
        Function for altering the power injection or consumption at a bus.
        """
        self.BusList[busnum - 1].pload += delta
```

```
    #Checks for overflow on all lines
#Ingrid    def checkOverflow(self):
    def checkFlow(self): #Ingrid
        """
        Function for checking for any overflows on any lin ein the system.
        """
#Ingrid        print('Checking for overflow on all lines:')
        found = 0
        zeroFList=[] #Ingrid
        smallFList=[] #Ingrid
        mediumFList=[] #Ingrid
        largeFList=[] #Ingrid
        overFList=[] #Ingrid
        for line in self.LineList:
#Ingrid            if line.ratea != 0:
            if line.ratea != 0 and line.ibstat==1: #Ingrid
                def uij(gij, bij, tetai, tetaj):
                    return gij * np.sin(tetai - tetaj) - bij * np.cos(tetai - tetaj)

                def tij(gij, bij, tetai, tetaj):
                    return gij * np.cos(tetai - tetaj) + bij * np.sin(tetai - tetaj)

                def bij(R, X):
                    return (1.0 / complex(R, X)).imag

                def gij(R, X):
                    return (1.0 / complex(R, X)).real

                ifr = line.fbus - 1
                itr = line.tbus - 1
                bsh = 0.0  # No shunts included so far
                teta1 = self.BusList[ifr].voang
                teta2 = self.BusList[itr].voang
                v1 = self.BusList[ifr].vomag
                v2 = self.BusList[itr].vomag
                b = bij(line.r, line.x)
                g = gij(line.r, line.x)

                Pfrom = g * v1 * v1 - v1 * v2 * tij(g, b, teta1, teta2)
                Pto = g * v2 * v2 - v1 * v2 * tij(g, b, teta2, teta1)
                Qfrom = -(b + bsh) * v1 * v1 - v1 * v2 * uij(g, b, teta1, teta2)
                Qto = -(b + bsh) * v2 * v2 - v1 * v2 * uij(g, b, teta2, teta1)
                Sfrom = math.sqrt(Pfrom**2 + Qfrom**2)
                Sto = math.sqrt(Pto ** 2 + Qto ** 2)
                tabS = line.ratea * percentS
                tabM = line.ratea * percentM
                tabL = line.ratea * percentL
                # 40% < line flow <= 60 %
                if (Sfrom > tabS and Sfrom <= tabM) or (Sto > tabS and Sto <= tabM):
                    smallFList.append(line)
                # 60% < line flow <= 80 %
                if (Sfrom > tabM and Sfrom <= tabL) or (Sto > tabM and Sto <= tabL):
                    mediumFList.append(line)
```

```
                # 80% < line flow <= 100 %
                if (Sfrom > tabL and Sfrom <= line.ratea) or (Sto > tabL and Sto
                ↪  <= line.ratea):
                    largeFList.append(line)
                if Sfrom > line.ratea or Sto > line.ratea:
#                    print('Overflow found at line between bus: ', line.tbus, '
↪   and ', line.fbus)
                    found += 1
                    overFList.append(line) # Ingrid, appending line objects.
                if Pfrom==0.0 or Qfrom==0.0 or Pto==0.0 or Qto==0.0:
                    #Ingrid; Use 'or' here and not 'and' to make sure that if
                    ↪  anything is making trouble here, we spot it.
                    zeroFList.append(line) # Ingrid, appending line objects.
                    print('No flow on the line from bus', line.fbus, 'to',
                    ↪  line.tbus)
        if found == 0:
            print('All line flows are within the limits.')
        return [zeroFList, smallFList, mediumFList, largeFList,
        ↪  overFList]#Ingrid

    #Get the potential voltage regulation at a bus
    def potential(self, bus):
        """
        Finds the maximum possible potential for voltage regulation at a bus.
        """
        # Get sensitivities
        sensP = bus.dVdP * (1.0 + bus.dPlossdP)
        sensQ = bus.dVdQ * (1.0 + bus.dQlossdQ)

        # Get available compensation
        compP = 0
        compQ = 0
        if bus.comp:
            compQ = self.SVCDroopCrtl(bus)  # Droop-based representation
        if bus.pv:
            pvobj = bus.pv
            if pvobj.cmode == 2:
                pvobj.qinj = self.PVDroopCrtl(bus) * bus.controlScale
            compP += pvobj.injPmax
            compQ += pvobj.injQmax
        if bus.battery:
            pvobj = bus.battery
            if pvobj.cmode == 2:
                pvobj.qinj = self.BatteryDroopCrtl(bus) * bus.controlScale
            compP += pvobj.injPmax
            compQ += pvobj.injQmax
        if bus.v2g:
            pvobj = bus.v2g
            if pvobj.cmode == 2:
                pvobj.qinj = self.V2GDroopCrtl(bus) * bus.controlScale
            compP += pvobj.injPmax
            compQ += pvobj.injQmax

        # Find the possible voltage regulation available
        vComp = sensP * compP + sensQ * compQ
        vComp = - vComp
        return vComp
```

```
    #Find out the needed change in power injection at a bus to correct a voltage mismatch
    def neededInjection(self, busnum, actOrReact=None):
        """
        The functions finds the needed power injection needed at a bus to get the voltage back within its limits.
        """
        bus = self.BusList[busnum - 1]
        deltaV = 0.0
        inj = 0.0
        typ = actOrReact
        sens = 0
        if typ == 'active':
            sens = bus.dVdP * (1.0 + bus.dPlossdP)
        elif typ == 'reactive':
            sens = bus.dVdQ * (1.0 + bus.dQlossdQ)
        increase = 0
        decrease = 0
        if bus.vomag > bus.vmax:
            deltaV = bus.vomag - bus.vmax
            print('The bus voltage at bus', bus.busnum, ' needs to be lowered by ', deltaV)
            inj = deltaV / sens
            if sens < 0:
                increase = 1
            if sens > 0:
                decrease = 1
        elif bus.vomag < bus.vmin:
            deltaV = bus.vmin - bus.vomag
            print('The bus voltage at bus ', bus.busnum, ' needs to be increased by ', deltaV)
            inj = deltaV / sens
            if sens < 0:
                decrease = 1
            if sens > 0:
                increase = 1
        else:
            print('The bus voltage at bus ', bus.busnum, ' is within its range')
        if increase:
            print(typ, ' power injection at bus ', bus.busnum, ' must be increased by ', abs(inj))
        if decrease:
            print(typ, ' power injection at bus ', bus.busnum, ' must be decreased by ', abs(inj))
        return inj


    def neededInjectionLine2(self, line):
        """
        Finds the needed active power injection needed at a bus in case of an overflow on a line.
        """
        lobj = None
        if type(line) is Line:
            lobj = line
        if type(line) is int:
            lobj = self.LineList[line]
```

```
        def getDelta(lobj1):
            def uij(gij, bij, tetai, tetaj):
                return gij * np.sin(tetai - tetaj) - bij * np.cos(tetai - tetaj)

            def tij(gij, bij, tetai, tetaj):
                return gij * np.cos(tetai - tetaj) + bij * np.sin(tetai - tetaj)

            ifr = lobj1.fbus - 1
            itr = lobj1.tbus - 1
            teta1 = self.BusList[ifr].voang
            teta2 = self.BusList[itr].voang
            v1 = self.BusList[ifr].vomag
            v2 = self.BusList[itr].vomag
            b = (1.0 / complex(lobj1.r, lobj1.x)).imag
            g = (1.0 / complex(lobj1.r, lobj1.x)).real

            Pfrom = g * v1 * v1 - v1 * v2 * tij(g, b, teta1, teta2)
            Qfrom = -b * v1 * v1 - v1 * v2 * uij(g, b, teta1, teta2)
            Sfrom1 = math.sqrt(Pfrom**2 + Qfrom**2)
            deltaS1 = Sfrom1 - lobj.ratea
            if deltaS1 ** 2 > Qfrom ** 2:                                  #Extra
            ↪   check to make it compile even if it is within its limit.
                neededP = math.sqrt(deltaS1 ** 2 - Qfrom ** 2)
            else:
                neededP = None
            return deltaS1, neededP

        deltaS, neededP = getDelta(lobj)
        if deltaS <= 0:
            print('Line flow between bus ', lobj.fbus, ' and ', lobj.tbus, ' is
            ↪   within limits.')
            return 0.0
        if deltaS > 0:
            print('Line flow on line between bus ', lobj.fbus, ' and ',
            ↪   lobj.tbus, ' must be lowered by ', deltaS)
            if neededP is None:
                print('The line flow cannot be corrected solely by active
                ↪   injection at bus ', lobj.tbus)
            else:
                print('Active injection at bus ', lobj.tbus, ' can be increased
                ↪   by ', neededP)
        return deltaS, neededP


    # Handle an overload
    def handleOverload(self, overloaded):
        """
        Function to handle an overload at one or several buses. Disconnects them,
↪   and tries to connect the reserve
        lines present in the system. Finds the reserve line that connects the
↪   most buses and results in the lowest
        losses.
        """
        self.disconnectBuses(overloaded)
        print('Trying different topologies to find a solution: \n')
        reserve = []
        connected = None
        for line in self.LineList:
```

```
            if line.reserve == 1:
                reserve.append(line)
        plossmin = 10000
        numbus = 0
        for line in reserve:
            self.connectLine2(line)
            self.config3()
            mesh = self.findtree()
            if mesh is None:
                self.config3()
                self.topology = dlf.mainstruct4()
                p1, q1, p2, q2 = self.accload(self.topology, self.BusList)
                connectedbuses = self.connectedBuses(self.topology)
                if len(connectedbuses) >= numbus:
                    if p2 < plossmin:
                        numbus = len(connectedbuses)
                        plossmin = p2
                        connected = line
            self.disconnectLine2(line)
        if plossmin < 10000:
            self.connectLine2(connected)
            self.config3()
            mesh = self.findtree()
            self.config3()
            self.topology = dlf.mainstruct4()
            print('\nNetwork was altered due to an overload at bus: ' +
            ↪   str([o.busnum for o in overloaded]) + '\n' +
                  'Network was altered by connecting line: ' +
                  ↪   str(self.LineList.index(connected)) + ' between bus: ' +
                  str(connected.tbus) + ' and ' + str(connected.fbus))
            top = self.connectedBuses(self.topology)
            print('Number of buses connected: ', len(top))
            self.resetBuses()
            print('New Load Flow Solution: \n')
            dlf.DistLF(epsilon=0.00001)
        if plossmin == 10000:
            print('No alternative topology could be found to alter the network
            ↪   and still have a radial network')

    # Display transmission line flows
    def dispFlow(self, fromLine=0, toLine=0, tpres=False, case=None, FLists=[]):
    ↪   #Ingrid, included case and FLists.
        """ Display the flow on the requested distribution lines
        """

        mainlist = []
        rowno = []

        def uij(gij, bij, tetai, tetaj):
            return gij * np.sin(tetai - tetaj) - bij * np.cos(tetai - tetaj)

        def tij(gij, bij, tetai, tetaj):
            return gij * np.cos(tetai - tetaj) + bij * np.sin(tetai - tetaj)

        def bij(R, X):
            return (1.0 / complex(R, X)).imag

        def gij(R, X):
```

```
            return (1.0 / complex(R, X)).real

        if toLine == 0:
            toLine = len(self.LineList)
#        if tpres:
#Ingrid            toLine = np.minimum(fromLine + 13, toLine)
            toLine = np.minimum(fromLine + 62, toLine) #Ingrid

        if fromLine < len(self.LineList):
            inum = fromLine
        else:
            print('Line :', fromLine, ' does not exist')
            return()

        for line in self.LineList[fromLine:toLine]:
            ifr = line.fbus - 1
            itr = line.tbus - 1
            bsh = 0.0  # No shunts included so far
            teta1 = self.BusList[ifr].voang
            teta2 = self.BusList[itr].voang
            v1 = self.BusList[ifr].vomag
            v2 = self.BusList[itr].vomag
            b = bij(line.r, line.x)
            g = gij(line.r, line.x)

            Pfrom = g * v1 * v1 - v1 * v2 * tij(g, b, teta1, teta2)
            Pto = g * v2 * v2 - v1 * v2 * tij(g, b, teta2, teta1)
            Qfrom = -(b + bsh) * v1 * v1 - v1 * v2 * uij(g, b, teta1, teta2)
            Qto = -(b + bsh) * v2 * v2 - v1 * v2 * uij(g, b, teta2, teta1)
            # Update structures
            line.flowfromP = Pfrom
            line.flowfromQ = Qfrom
            line.flowtoP = Pto
            line.flowtoQ = Qto

            if not tpres:
                print(' FromBus :', '{:4.0f}'.format(ifr + 1), ' ToBus :',
                 ↪  '{:4.0f}'.format(itr + 1),
                      ' Pfrom :', '{:7.4f}'.format(Pfrom), ' Qfrom : ',
                       ↪  '{:7.4f}'.format(Qfrom),
                      ' Pto :', '{:7.4f}'.format(Pto), ' Qto :',
                       ↪  '{:7.4f}'.format(Qto))

#Ingrid            sublist = [ifr + 1, itr + 1, '{:7.4f}'.format(Pfrom),
↪   '{:7.4f}'.format(Qfrom),
#Ingrid                       '{:7.4f}'.format(Pto), '{:7.4f}'.format(Qfrom)]
            Sfrom = math.sqrt(Pfrom**2 + Qfrom**2)
            Sto = math.sqrt(Pto ** 2 + Qto ** 2)
            sublist = [ifr + 1, itr + 1,
#            sublist = [str(ifr + 1)+'-'+str(itr + 1),
#                    '{:7.4f}'.format(Sfrom),
#                    '{:7.4f}'.format(Sto)]#,
                    '{:7.4f}'.format(Pfrom),
                    '{:7.4f}'.format(Pto),
                    '{:7.4f}'.format(Qfrom),
                    '{:7.4f}'.format(Qto)] #Ingrid, changed the last Q from Qfrom
                     ↪  to Qto. Must have been a copy-paste error.
            mainlist.append(sublist)
```

```
#Ingrid            rowno.append('Line ' + str(inum))
            rowno.append('L' + str(inum+1)) #Ingrid
            inum += 1

        if tpres:
            title = 'Transmission line flow'
#Ingrid            colind = ['FromBus', 'ToBus', 'Pfrom', 'Qfrom', 'Pto', 'Qto']
#            colind = ['fromBus', 'toBus', '$S_{from}$', '$S_{to}$']#, '$P_{to}$', '$Q_{to}$'] #Ingrid
#            colind = ['Buses', '$P_{from}$', '$P_{to}$', '$Q_{from}$', '$Q_{to}$'] #Ingrid
            colind = ['fromBus', 'toBus', '$P_{from}$', '$P_{to}$', '$Q_{from}$', '$Q_{to}$'] #Ingrid
#Ingrid            self.tableplot(mainlist, title, colind, rowno, columncol=[], rowcol=[], colw=[], case=case) #Ingrid, included case and colw.
            self.tableplot(mainlist, title, colind, rowno, case, FLists) #Ingrid, included case and FLists.


    # Conduct a distribution system load flow based on FBS
    def DistLF(self, epsilon=0.0001):
        """ Solves the distribution load flow until the convergence criteria is met for all buses.
        The two first steps are to set up additions topology information and to build the main structure
        Next,it is switched between forward sweeps(Voltage updates) and backward sweeps(load update and loss calcuation)
        """

# Flat start option has to be considered
        self.flatStart()

        diff = 10
        iloop = 0
        while diff > epsilon:
            p1, q1, p2, q2 = self.accload(self.topology, self.BusList)
            print('Iter: ', iloop + 1, 'Pload:', '{:7.4f}'.format(p1), 'Qload:', '{:7.4f}'.format(q1),
                  'Ploss:', '{:7.4f}'.format(p2), 'Qloss:', '{:7.4f}'.format(q2))
            oldVs = []
            for i in range(0, len(self.BusList)):
                oldVs.append(self.BusList[i].vomag)
            self.UpdateVolt(self.topology, self.BusList)
            newVs = []
            iloop += 1
            if iloop > 15:
                print('Convergence could not be reached.')
                return [] #Ingrid. Return empty list.
            diffs = []
            for i in range(0, len(self.BusList)):
                newVs.append(self.BusList[i].vomag)
                diffs.append(abs(oldVs[i] - newVs[i]))
            diff = max(diffs)
#        overload = self.checkOverLoad()
#        if len(overload) > 0:
#            self.handleOverload(overload)
        print("****** Load flow completed in ", iloop, " iterations ******")
#Ingrid        print('\n', "****** Load flow completed in ", iloop, " iterations ******", '\n')
```

```
        sublist = ['{:7.4f}'.format(p1), '{:7.4f}'.format(q1), '{:7.4f}'.format(p2), '{:7.4f}'.format(q2)]#Ingrid. Return total loads and total losses in string format.
        return sublist #Ingrid. Added this line in order to tabulate each case's total power load and total loss.



    # Visit all nodes in the reverse list.
    def BackwardSearch(self, topologyList):
        """ Visit all the nodes in a backward approach and prints the Bus name
        """
        for x in reversed(topologyList):
            if len(x) > 1:
                print('Bus' + str(x[0].busnum))
                iloop = 1
                while iloop < len(x):  # Do for all branches of a bus
                    self.BackwardSearch(x[iloop])
                    iloop += 1
            else:
                print('Bus' + str(x[0].busnum))

    # Visit all nodes in the forward list.
    def ForwardSearch(self, topologyList):
        """Visit all nodes in a forward approach and prints the Bus name
        """
        for x in topologyList:
            if len(x) > 1:
                print('Bus' + str(x[0].busnum))
                iloop = 1
                while iloop < len(x):  # Do for all branches of a bus
                    self.ForwardSearch(x[iloop])
                    iloop += 1
            else:
                print('Bus' + str(x[0].busnum))



    # Visit all nodes in the forward list.
    def AddNodes(self, G=0, topologyList=[], tagBus=[]):
        """Visit all nodes in a forward approach and build the graphic representation
        """
        def tagNode(G=0, node=0, tagBus=[]):
            # The colors are set at top of this .py-file.
            # tagBus = [CINELDI124feeders, batteries, V2Gs, ferry, charging, [feeder]] + VLists
            # VLists = [zero, under, medium, large, over]
            S = 20
            M = 30
            L = 40
            XL = 50

            busnr = node.busnum #To be uses many times below.

            #-------label-----------------color----size--
            tag = [['B' + str(busnr) + '\n',buses,S],
```

```
                   ['***Bck feeder***',     supply,   M],
                   ['***Battery***',        charge,   M],
                   ['V2G',                  charge,   M],
                   ['Ferry',                charge,   M],
                   ['---charging---',       charge,   M],
                   ['---supplying---',      supply,  XL],
                   ['Zero Volt',            zero,    XL],
                   ['Low Volt',             small,   XL],
                   ['Medium Volt',          medium,   S],
                   ['Large Volt',           large,    S],
                   ['Over Volt',            over,    XL]]
#                   ['LEC',                  lec,      L]]

            # If multiple labels per node, display them all:
            indextags=[]
            indextags.append(0) # Make sure the node's bus number is tagged.
            for taglist in tagBus:
                for bobj in taglist:
                    if busnr==bobj.busnum: #node is eaten by tagNode (currently defined).
                        i = tagBus.index(taglist) + 1
                        if i not in indextags: #not get same label twice or more.
                            indextags.append(i)
                        break
            text=''
            for ind in indextags:
                label = tag[ind][0]
                if ind==1 and busnr==1:
                    label = '***Main feeder***' # Overwrite if node is main feeder, because CINELDI124 has its main feeder at bus1, and its back up feeders at buses, 36, 62 and 88.
                text += label + '\n'

            # Tag node with attributes:
            i = indextags[-1] # The last tag gets the highest color priority.
            for ind in indextags:
                if tag[ind][1]==charge: #Make sure that Ferry, V2G etc. is visible on the tree.
                    i=ind
                    break
            G.add_node(busnr,
                        label = text,
                        color = tag[i][1],
                        size  = tag[i][2])

        # Build network of nodes:
        for x in topologyList:
            if len(x) > 1:
#                print('Bus' + str(x[0].busnum))
                tagNode(G, x[0], tagBus)

                iloop = 1
                while iloop < len(x):  # Do for all branches of a bus
                    self.AddNodes(G, x[iloop], tagBus)
                    iloop += 1
            else:
#                print('Bus' + str(x[0].busnum))
```

```
                tagNode(G, x[0], tagBus)



   # Visit all nodes in the reverse list.
    def ConnectNodes(self, G=0, topologyList=[], tagLine=[]):
        """ Visit all the nodes in a backward approach and prints the Bus name
        """
        def tagEdge(G=0, toLine=0, toNode=0, tagLine=[]):
            # The colors are set at top of this .py-file.
            #tagLine = [zero, small, medium, large, over]
            value = 500
            #--------label----------color-------value-arrows---
            tag = [['',             lines,      value, True],
                   ['ZERO FLOW',    zero,       value, False],
                   ['',            small,      value, True],
                   ['',           medium,     value, True],
                   ['',          large,      value, True],
                   ['OVERLOADED',   over,       value, True]]

            # Choose the appropriate tag:
            i=0
            for taglist in tagLine:
                for lobj in taglist:
                    if lobj.fbus == toLine.fbus and lobj.tbus == toNode.busnum:
                        i = tagLine.index(taglist) + 1
                        break

            # Tag line with attributes:
            linenr = toLine.linenum
            text = 'L' + str(linenr) + '\n' + tag[i][0]
            G.add_edge(toLine.fbus,
                    toNode.busnum,
                    label  = text,
                    color  = tag[i][1],
                    value  = tag[i][2],
                    arrows = tag[i][3])

        # Connect nodes:
        for x in reversed(topologyList):
            if len(x) > 1:
#                print('Bus' + str(x[0].busnum))
                if x[0].toline:
                    tagEdge(G, x[0].toline, x[0], tagLine)

                iloop = 1
                while iloop < len(x):  # Do for all branches of a bus
                    self.ConnectNodes(G, x[iloop], tagLine)
                    iloop += 1
            else:
#                print('Bus' + str(x[0].busnum))
                if x[0].toline:
                    tagEdge(G, x[0].toline, x[0], tagLine)



    def dispTree(self, topologyList=[], tagBus=[], tagLine=[], filename=None):
```

```
        """
        Builds and displays the graph as a HTML-file

        """
        G = nx.DiGraph()
        self.AddNodes(G, topologyList, tagBus)
        self.ConnectNodes(G, topologyList, tagLine)
        nx.draw(G, with_labels = True)
        nt = Network('2000px', '2000px', directed=True, layout="Hierarchcal")
        nt.from_nx(G)

        # Create hyperlink where network is displayed:
        nt.show(filename+'.html')



    # Calculations the load for the actual voltage at the bus
    def getload(self, busobj):
        """ Calculates the net voltage corrected load at the bus - currently a
        ↪  simple ZIP model is applied.
        Input: The busobject
        Returns: pLoadAct, qLoadAct
        """
        #        if busobj.vset > 0:
        #            self.voltCrtl(busobj)
        qmod = 0.0
        pmod = 0.0
        # Include all possible local sources (SVC/Statcom, PV, Battery and V2G)
        if busobj.comp:
            qmod = self.SVCDroopCrtl(busobj)  # Droop-based representation
        if busobj.pv:
            pvobj = busobj.pv
            if pvobj.cmode == 2:
                pvobj.qinj = self.PVDroopCrtl(busobj) * busobj.controlScale
            pmod += pvobj.pinj
            qmod += pvobj.qinj
        if busobj.battery:
            pvobj = busobj.battery
            if pvobj.cmode == 2:
                pvobj.qinj = self.BatteryDroopCrtl(busobj) * busobj.controlScale
            pvobj = busobj.battery
            pmod += pvobj.pinj
            qmod += pvobj.qinj
        if busobj.v2g:
            pvobj = busobj.v2g
            if pvobj.cmode == 2:
                pvobj.qinj = self.V2GDroopCrtl(busobj) * busobj.controlScale
            pmod += pvobj.pinj
            qmod += pvobj.qinj
        # Find the net load at the node (Note: load - injection)
        pLoadAct = busobj.pload * (
                busobj.ZIP[0] * busobj.vomag ** 2 + busobj.ZIP[1] * busobj.vomag
                ↪  + busobj.ZIP[2]) - pmod
        qLoadAct = busobj.qload * (
                busobj.ZIP[0] * busobj.vomag ** 2 + busobj.ZIP[1] * busobj.vomag
                ↪  + busobj.ZIP[2]) - qmod
        dPdV = busobj.pload * (busobj.ZIP[0] * 2 * busobj.vomag + busobj.ZIP[1])
        dQdV = busobj.qload * (busobj.ZIP[0] * 2 * busobj.vomag + busobj.ZIP[1])
```

```
        return pLoadAct, qLoadAct, dPdV, dQdV

    def voltCrtl(self, busobj, mode='Reactive'):
        """ Changes the net injection at voltage controlled buses
                Input: The busobject
                mode - Control mode ('Active', 'Reactive', 'Both' - default =
↪   'Reactive')
                Returns: pLoadAct, qLoadAct
                """
        if busobj.vset > 0 and busobj.vomag < 1.0:
            if np.abs(busobj.vomag - busobj.vset) > 0.0002:
                if mode == 'Active':
                    deltap = (busobj.vset - busobj.vomag) / (busobj.dVdP * (1 +
                     ↪  busobj.dPlossdP))
                    busobj.pload += deltap
                    print('Load corr (Active): ', busobj.busnum, deltap,
                     ↪  busobj.pload)
                elif mode == 'Reactive':
                    deltaq = (busobj.vset - busobj.vomag) / (busobj.dVdQ * (1 +
                     ↪  busobj.dQlossdQ))
                    busobj.qload += deltaq
                    print('Load corr (Reactive): ', busobj.busnum, deltaq,
                     ↪  busobj.qload)

    def PVDroopCrtl(self, busobj):
        """Calculates the PV/converter contribution to voltage control"""
        pvobj = busobj.pv
        if pvobj.stat:
            qsens = busobj.dVdQ * (1.0 + busobj.dQlossdQ)
            if qsens:
                a = 1.0
                b = -(pvobj.vprev + pvobj.slopeQ / qsens)
                c = pvobj.slopeQ / qsens * busobj.vomag
                v = (-b + np.sqrt(b ** 2 - 4 * a * c)) / 2.0
                v2 = (-b - np.sqrt(b ** 2 - 4 * a * c)) / 2.0
                # print('v1 :', v, '   v2 : ', v2)
            else:
                v = pvobj.vprev
            #        v = (busobj.vomag - qsens*svcobj.vprev/svcobj.slopeQ)/(1.0 -
             ↪  qsens/svcobj.slopeQ)
            Qc = -1.0 / pvobj.slopeQ * v * (v - pvobj.vref)
            pvobj.vprev = v
            print(busobj.busname, '     Volt: ', v, '   Qinj = ', Qc)
            pvobj.qinj = Qc
            return Qc

    def V2GDroopCrtl(self, busobj):
        """Calculates the PV/converter contribution to voltage control"""
        v2gobj = busobj.v2g
        if v2gobj.stat:
            qsens = busobj.dVdQ * (1.0 + busobj.dQlossdQ)
            if qsens:
                a = 1.0
#Ingrid                b = -(v2gobj.vprev + v2gobj.slopeQ / qsens)
#Ingrid                c = v2gobj.slopeQ / qsens * busobj.vomag
                temp = qsens / v2gobj.slopeQ #Ingrid
                b = -(v2gobj.vprev + temp) #Ingrid
                c = temp * busobj.vomag #Ingrid
```

```
                v = (-b + np.sqrt(b ** 2 - 4 * a * c)) / 2.0
                v2 = (-b - np.sqrt(b ** 2 - 4 * a * c)) / 2.0
            # print('v1 :', v, '   v2 : ', v2)
            else:
                v = v2gobj.vprev
            #        v = (busobj.vomag - qsens*svcobj.vprev/svcobj.slopeQ)/(1.0 - qsens/svcobj.slopeQ)
#Ingrid            Qc = -1.0 / v2gobj.slopeQ * v * (v - v2gobj.vref)
            Qc = -1.0 * v2gobj.slopeQ * v * (v - v2gobj.vref) #Ingrid
            v2gobj.vprev = v
            print(busobj.busname, '     Volt: ', v, '   Qinj = ', Qc)
            v2gobj.qinj = Qc
            return Qc

    def BatteryDroopCrtl(self, busobj):
        """Calculates the PV/converter contribution to voltage control"""
        batobj = busobj.battery
        if batobj.stat:
            qsens = busobj.dVdQ * (1.0 + busobj.dQlossdQ)
            if qsens:
                a = 1.0
#Ingrid                b = -(batobj.vprev + batobj.slopeQ / qsens)
#Ingrid                c = batobj.slopeQ / qsens * busobj.vomag
                temp = qsens / batobj.slopeQ #Ingrid
                b = -(batobj.vprev + temp) #Ingrid
                c = temp * busobj.vomag #Ingrid
                v = (-b + np.sqrt(b ** 2 - 4 * a * c)) / 2.0
                v2 = (-b - np.sqrt(b ** 2 - 4 * a * c)) / 2.0
            # print('v1 :', v, '   v2 : ', v2)
            else:
                v = batobj.vprev
            #        v = (busobj.vomag - qsens*svcobj.vprev/svcobj.slopeQ)/(1.0 - qsens/svcobj.slopeQ)
#Ingrid            Qc = -1.0 / batobj.slopeQ * v * (v - batobj.vref)
            Qc = -1.0 * batobj.slopeQ * v * (v - batobj.vref) #Ingrid
            batobj.vprev = v
            print(busobj.busname, '     Volt: ', v, '   Qinj = ', Qc)
            batobj.qinj = Qc
            return Qc

    def SVCDroopCrtl(self, busobj):
        """Calculates the SVC contribution to voltage control"""
        svcobj = busobj.comp
        if svcobj.stat:
            qsens = busobj.dVdQ * (1.0 + busobj.dQlossdQ)
            if qsens:
                a = 1.0
                b = -(svcobj.vprev + svcobj.slopeQ / qsens)
                c = svcobj.slopeQ / qsens * busobj.vomag
                v = (-b + np.sqrt(b ** 2 - 4 * a * c)) / 2.0
                v2 = (-b - np.sqrt(b ** 2 - 4 * a * c)) / 2.0
                print('v1 :', v, '   v2 : ', v2)
            else:
                v = svcobj.vprev
            #        v = (busobj.vomag - qsens*svcobj.vprev/svcobj.slopeQ)/(1.0 - qsens/svcobj.slopeQ)
            Qc = -1.0 / svcobj.slopeQ * v * (v - svcobj.vref)
            svcobj.vprev = v
```

```
            print(busobj.busname, '     Volt: ', v, '   Qinj = ', Qc)
            svcobj.qinj = Qc
            return Qc

    def SVCCrtl2(self, busobj):
        """Calculates the SVC contribution to voltage control"""
        svcobj = busobj.comp
        if svcobj.stat:
            qsens = busobj.dVdQ * (1.0 + busobj.dQlossdQ)
            v = (busobj.vomag - qsens * svcobj.vprev / svcobj.slopeQ) / (1.0 -
             ↪  qsens / svcobj.slopeQ)
            Qc = -1.0 / svcobj.slopeQ * (v - svcobj.vref)
            svcobj.vprev = v
            print(busobj.busname, '     Volt: ', v, '   Qinj = ', Qc)
            svcobj.qinj = Qc
            return Qc

    # Calculate the accumulated load and losses starting on the last node
    def accload(self, topologyList, BusList):
        """Calculates the accumulated downstream active and reactive load at all
         ↪  buses
        and calculates the active and reactive losses of lines and make an
↪   accumulated equivalent load at the buses
        """
        pl1 = 0.0
        ql1 = 0.0
        ploss1 = 0.0
        qloss1 = 0.0

        for x in reversed(topologyList):  # Start on last node
            if len(x) > 1:
                iloop = 1
                while iloop < len(x):  # Do for all branches at a bus
                    pl2, ql2, ploss2, qloss2 = self.accload(x[iloop], BusList)
                    pl1 += pl2  # Add accumulated powers and losses in a branch
                     ↪  to the node where the brancing accurs.
                    ql1 += ql2
                    ploss1 += ploss2
                    qloss1 += qloss2
                    iloop += 1
                pla, qla, dPdV1, dQdV1 = self.getload(x[0])  # Add local loads
                pl1 += pla  # Add local loads
                ql1 += qla
                x[0].ploadds = pl1  # Add accumulated descriptions to the
                 ↪  branching node
                x[0].qloadds = ql1
                x[0].pblossds = ploss1
                x[0].qblossds = qloss1
                if pl1 != 0:
                    x[0].dPdV = (x[0].dPdV * (pl1 - pla) + dPdV1 * pla) / pl1
                if ql1 != 0:
                    x[0].dQdV = (x[0].dQdV * (ql1 - qla) + dQdV1 * qla) / ql1
                if x[0].toline:  # Follow the next node in the main path
                    lobj = x[0].toline
                    if lobj.ibstat:
                        ifr = lobj.fbus
                        itr = lobj.tbus
```

```
                        pto = x[0].ploadds + x[0].pblossds  # Find the flow to the downstream bus
                        qto = x[0].qloadds + x[0].qblossds
                        lobj.ploss = lobj.r * (pto ** 2 + qto ** 2) / x[
                            0].vomag ** 2  # Estimate the losses of the branch
                        lobj.qloss = lobj.x * (pto ** 2 + qto ** 2) / x[0].vomag ** 2
                        ploss1 += lobj.ploss
                        qloss1 += lobj.qloss
                        x[0].pblossds = ploss1  # Add the losses to the downstream bus
                        x[0].qblossds = qloss1

            else:  # No branching at the bus
                #                 pl1 += x[0].pload
                #                 ql1 += x[0].qload
                pla, qla, dPdV1, dQdV1 = self.getload(x[0])
                pl1 += pla  # Add local loads
                ql1 += qla
                x[0].ploadds = pl1
                x[0].qloadds = ql1
                if pl1 != 0:
                    x[0].dPdV = (x[0].dPdV * (pl1 - pla) + dPdV1 * pla) / pl1
                if ql1 != 0:
                    x[0].dQdV = (x[0].dQdV * (ql1 - qla) + dQdV1 * qla) / ql1
                if x[0].toline:
                    lobj = x[0].toline
                    if lobj.ibstat:
                        ifr = lobj.fbus
                        itr = lobj.tbus
                        pto = x[0].ploadds + ploss1
                        qto = x[0].qloadds + qloss1
                        lobj.ploss = lobj.r * (pto ** 2 + qto ** 2) / x[0].vomag ** 2
                        lobj.qloss = lobj.x * (pto ** 2 + qto ** 2) / x[0].vomag ** 2
                        ploss1 += lobj.ploss
                        qloss1 += lobj.qloss
                        x[0].pblossds = ploss1
                        x[0].qblossds = qloss1

        return pl1, ql1, ploss1, qloss1  # Return the accumulated loads and losses from the current branch

    # Update the control scaling factors
    def UpdateControl(self, BusList):
        """ Updates the scaling factors used for Voltage and Minimum loss purposes
            Identifies number of control units on adjacent buses and updates the scaling used to improve convergence

            May be extended later
        """

        iloop = 0
        while iloop < len(BusList):
            iunit = 0
            inext = BusList[iloop].tolinelist
```

```
            # print(len(inext))
            if len(inext) > 1:
                for iloop2 in inext:  # Find the number of buses
                    itr = iloop2.tbus
                    if BusList[itr - 1].iloss == 1:
                        iunit += 1
                        print(BusList[itr - 1].busname)
                if iunit > 1:  # Update the scaling factors
                    for iloop2 in inext:
                        itr = iloop2.tbus
                        if BusList[itr - 1].iloss == 1:
                            BusList[
                                itr - 1].controlScale = 1.2 / iunit  # Use: 1.0
                                ↪   (default), 0.6, 0.4 and 0.3 depending on the
                                ↪   number of control buses

            iloop += 1
        # End

    # Update the voltage profile starting on the top node
    def UpdateVolt(self, topologyList, BusList):
        """Update the voltage profile based on the accumulated load on each bus
        """

        # Function for calculating the voltages and sensitivities in the single
        ↪   phase case (modified sensitivity calculation)
        def nodeVoltSensSPv2(BusList, ifr, itr, tline, obj):
            """
            Calculate the node voltages and sensitivities in the single phase
↪   case - a more accurate sensitivity calculation (had just minor impact)
            :param BusList:
            :param ifr:
            :param itr:
            :param tline:
            :param obj:
            :return:
            """

            vk2 = BusList[ifr].vomag ** 2
            tpload = obj[0].ploadds + obj[0].pblossds  # Find the accumulated
            ↪   loads and losses flowing on the branch
            tqload = obj[0].qloadds + obj[0].qblossds
            # Voltage calculation
            term2 = 2 * (tpload * tline.r + tqload * tline.x)
            term3 = (tpload ** 2 + tqload ** 2) * (tline.r ** 2 + tline.x ** 2) /
            ↪   BusList[ifr].vomag ** 2
            BusList[itr].vomag = np.sqrt(
                vk2 - term2 + term3)  # Update the bus voltage magnitude on the
                ↪   down-stream bus
            # Calculate the sensitivities for changing the load
            #   dvdp = (-tline.r + tpload * (tline.r ** 2 + tline.x ** 2) /
            ↪   BusList[ifr].vomag ** 2) / BusList[
            #       itr].vomag

            dqdp = (2 * tline.x * tpload / BusList[itr].vomag ** 2) * (
                    1 + 2 * tline.x * tqload / BusList[
                itr].vomag ** 2)  # The relation between the chang in q for a
                ↪   change in p - simplified version to get a better dvdp
```

```
            dvdp = (-tline.r - tline.x * dqdp + (tpload + tqload * dqdp) * (tline.r ** 2 + tline.x ** 2) / BusList[
                ifr].vomag ** 2) / BusList[
                       itr].vomag

            dpdq = (2 * tline.r * tqload / BusList[itr].vomag ** 2) * (
                    1 + 2 * tline.r * tpload / BusList[
                itr].vomag ** 2)  # The relation between the change in p for a change in q - simplified version
            #  dvdq = (-tline.x + tqload * (tline.r ** 2 + tline.x ** 2) / BusList[ifr].vomag ** 2) / BusList[
            #     itr].vomag
            dvdq = (-tline.x - tline.r * dpdq + (tqload + tpload * dpdq) * (tline.r ** 2 + tline.x ** 2) / BusList[
                ifr].vomag ** 2) / BusList[
                       itr].vomag

            # dqdp = (2 * tline.x * tpload / BusList[itr].vomag ** 2) * (
            #         1 + 2 * tline.x * tqload / BusList[
            #     itr].vomag ** 2)  # The relation between the chang in q for a change in p
            dqdp = ((2 * tline.x * tqload + 2 * tline.x * tpload * dpdq) * BusList[itr].vomag ** 2 - (
                    tline.x * tpload ** 2 + tline.x * tqload ** 2) * 2 * BusList[itr].vomag * dvdp) / BusList[
                       itr].vomag ** 4

            dpdq = ((2 * tline.r * tqload + 2 * tline.r * tpload * dqdp) * BusList[itr].vomag ** 2 - (
                    tline.r * tpload ** 2 + tline.r * tqload ** 2) * 2 * BusList[itr].vomag * dvdq) / BusList[
                       itr].vomag ** 4

            #  dpldp = (2 * tline.r * tpload / BusList[itr].vomag ** 2) * (
            #          1 + 2 * tline.x * tqload / BusList[itr].vomag ** 2)  # Change in losses for a change in p
            dpldp = ((2 * tline.r * tpload + 2 * tline.r * tqload * dqdp) * BusList[itr].vomag ** 2 - (
                    tline.r * tpload ** 2 + tline.r * tqload ** 2) * 2 * BusList[itr].vomag * dvdp) / BusList[
                        itr].vomag ** 4

            BusList[itr].dVdP = BusList[ifr].dVdP + dvdp + dvdq * dqdp
            BusList[itr].dVdQ = BusList[ifr].dVdQ + dvdq + dvdp * dpdq
            # Calculate sensitivities for change in losses
            BusList[itr].dPlossdP = BusList[ifr].dPlossdP + dpldp
            BusList[itr].dPlossdQ = BusList[ifr].dPlossdQ + dpdq
            BusList[itr].dQlossdP = BusList[ifr].dQlossdP + dqdp
            #                    BusList[itr].dQlossdQ = BusList[ifr].dQlossdQ + (2 * tline.x * tqload/BusList[itr].vomag**2) * (1 + 2 * tline.r * tpload/BusList[itr].vomag**2)
            BusList[itr].dQlossdQ = BusList[ifr].dQlossdQ + 2 * tline.x * tqload / BusList[
                itr].vomag ** 2 + 2 * tline.x * tpload * BusList[itr].dPlossdQ / BusList[itr].vomag ** 2
            # Calculate the second-order derivatives
            if tqload == 0:
```

```
            term1q = 0
        else:
            term1q = dpdq / tqload
        BusList[itr].dP2lossdQ2 = BusList[ifr].dP2lossdQ2 + term1q + (
                2 * tline.r * tqload / BusList[itr].vomag ** 2) * 2 * tline.r
                ↪   * dpdq / BusList[itr].vomag ** 2

        if tpload == 0:
            term1p = 0
        else:
            term1p = dpldp / tpload
        BusList[itr].dP2lossdP2 = BusList[ifr].dP2lossdQ2 + term1p + (
                2 * tline.r * tpload / BusList[itr].vomag ** 2) * 2 * tline.x
                ↪   * dqdp / BusList[itr].vomag ** 2
        # Estimate the required injection to reach minimum loss
        BusList[itr].lossRatioQ = BusList[itr].dPlossdQ /
        ↪   BusList[itr].dP2lossdQ2  # Check this one
        BusList[itr].lossRatioP = BusList[itr].dPlossdP /
        ↪   BusList[itr].dP2lossdP2

        # Update the voltage for the purpose of loss minimization - adjust
        ↪   the sensitivity acording to the chosen step.
        if BusList[itr].iloss:
            #    if np.abs(BusList[itr].dPlossdQ) >= 1.0 / BusList[
            #        itr].pqcostRatio:  # Equivalent to that the dP cost more
            ↪   than pqcostRatio times dQ
            qcomp = BusList[itr].dPlossdQ / (
                    BusList[itr].dP2lossdQ2 - 1.0)  # Estimate the
                    ↪   toerethically required adjustment

            # BusList[itr].dPlossdQ = 0.0 # In general case we should find
            ↪   better solution

            # Assign the correction to the right source and scale according
            ↪   to the choosen strategy
            if BusList[itr].pv:
                pvobj = BusList[itr].pv
                if pvobj.cmode == 1:  # Update only ot the cmode = 1 - NB
                ↪   Other objects may be added under this section when iloss
                ↪   = 1
                    pvobj.qinj += qcomp * BusList[itr].controlScale
                    BusList[itr].dPlossdQ = 0.0  # In general case we should
                    ↪   find better solution
            if BusList[itr].battery:
                pvobj = BusList[itr].battery
                if pvobj.cmode == 1:  # Update only ot the cmode = 1 - NB
                ↪   Other objects may be added under this section when iloss
                ↪   = 1
                    pvobj.qinj += qcomp * BusList[itr].controlScale
                    BusList[itr].dPlossdQ = 0.0  # In general case we should
                    ↪   find better solution
            if BusList[itr].v2g:
                pvobj = BusList[itr].v2g
                if pvobj.cmode == 1:  # Update only ot the cmode = 1 - NB
                ↪   Other objects may be added under this section when iloss
                ↪   = 1
                    pvobj.qinj += qcomp * BusList[itr].controlScale
```

```
                        BusList[itr].dPlossdQ = 0.0  # In general case we should find better solution

            # Voltage angle calculation
            busvoltreal = BusList[ifr].vomag - (tpload * tline.r + tqload * tline.x) / BusList[ifr].vomag
            busvoltimag = (tqload * tline.r - tpload * tline.x) / BusList[ifr].vomag
            BusList[itr].voang = BusList[ifr].voang + np.arctan2(busvoltimag, busvoltreal)  # Update voltage angles
            return

        #  End

        for obj in topologyList:
            if len(obj) > 1:

                if obj[0].toline:
                    tline = obj[0].toline
                    ifr = tline.fbus - 1
                    itr = tline.tbus - 1

                    # Update voltages and sensitivities Single Phase
                    nodeVoltSensSPv2(BusList, ifr, itr, tline, obj)

                iloop = 1
                while iloop < len(obj):  # Update voltages along the branches
                    self.UpdateVolt(obj[iloop], BusList)
                    iloop += 1
            else:  # Continue along the current path
                if obj[0].toline:
                    tline = obj[0].toline
                    ifr = tline.fbus - 1
                    itr = tline.tbus - 1

                    # Update voltages and sensitivities Single Phase
                    nodeVoltSensSPv2(BusList, ifr, itr, tline, obj)

    # Estimate the losses of each line based on voltage level and accumulated flow
    def lossEstimate(self, busobjects, lineobjects):
        """Estimates the losses of each line based on voltage level and accumulated flow
        """
        for lobj in reversed(lineobjects):
            ifr = lobj.fbus - 1
            itr = lobj.tbus - 1
            pto = busobjects[itr].ploadds
            qto = busobjects[itr].qloadds
            lobj.ploss = lobj.r * (pto ** 2 + qto ** 2) / busobjects[itr].vomag ** 2
            lobj.qloss = lobj.x * (pto ** 2 + qto ** 2) / busobjects[itr].vomag ** 2
            busobjects[ifr].ploadds += lobj.ploss
            busobjects[ifr].qloadds += lobj.qloss

    # Display the voltages.
```

```
    def dispVolt(self, fromBus=0, toBus=0, tpres=False, case=None, VLists=[]):
     ↪  #Ingrid, included case.
        """
        Desc:    Display voltages at all buses
        Input:   tpres= False (Display in tableformat if True)
                 fromBus and toBus defines the block, If tpres=True, it will
 ↪  display 13 lines from fromBus
        Returns: None
        """
        mainlist = []
        rowno = []
        if toBus == 0:
            toBus = len(self.BusList)
#        if tpres:
#Ingrid            toBus = np.minimum(fromBus + 13, toBus)
            toBus = np.minimum(fromBus + 62, toBus)#Ingrid

        iloop = fromBus
        print(' ')
        while iloop < toBus:
            oref = self.BusList[iloop]
            if not tpres:
                print(' Bus no :', '{:4.0f}'.format(oref.busnum),
                      ' Vmag :', '{:7.5f}'.format(oref.vomag),
                      ' Theta :', '{:7.5f}'.format(oref.voang * 180 / np.pi))
            # Prepare for graphics presentation
#Ingrid            sublist = ['{:4.0f}'.format(oref.busnum),
#Ingrid                       '{:7.5f}'.format(oref.vomag),
#Ingrid                       '{:7.5f}'.format(oref.voang * 180 / np.pi)]
            sublist = ['{:7.5f}'.format(oref.vomag),
                       '{:7.5f}'.format(oref.voang * 180 / np.pi)]
            mainlist.append(sublist)
#Ingrid            rowno.append('Bus ' + str(iloop + 1))
            rowno.append('B' + str(iloop + 1)) #Ingrid
            iloop += 1
        # Present table
        if tpres:
            title = 'Bus Voltages'
#Ingrid            colind = ['Bus nr', 'Vmag', 'Theta']
#Ingrid            colw = [0.12, 0.22, 0.22, 0.22, 0.22]
            colind = ['$V_{mag}$', '$\Theta_{V}$'] #Ingrid
            self.tableplot(mainlist, title, colind, rowno, case, VLists) #Ingrid,
             ↪  included case and VLists.


    # Display the voltages.
    def dispLowVolt(self, fromBus=0, toBus=0, tpres=False, vmax=1.1):
        """
        Desc:    Display voltages at all buses below or equal to the limit vmax
        Input:   tpres= False (Display in tableformat if True)
                 fromBus and toBus defines the block, If tpres=True, it will
 ↪  display 13 lines from fromBus
                 vmax = Upper voltage limit (default 1.1 pu)
        Returns: None
        """
        mainlist = []
        rowno = []
        if toBus == 0:
```

```
            toBus = len(self.BusList)
        #        if tpres:
        #            toBus = np.minimum(fromBus + 13, toBus)

        if fromBus < len(self.BusList): # Check legal range
            iloop = fromBus
        else:
            print(' Bus :', fromBus, ' does not exist')
            return()

        print(' ')
        while iloop < toBus:
            oref = self.BusList[iloop]
            if oref.vomag <= vmax:
                if not tpres:
                    print(' Bus no :', '{:4.0f}'.format(oref.busnum),
                          ' Vmag :', '{:7.5f}'.format(oref.vomag),
                          ' Theta :', '{:7.5f}'.format(oref.voang * 180 / np.pi))
                # Prepare for graphics presentation
                sublist = ['{:4.0f}'.format(oref.busnum),
                           '{:7.5f}'.format(oref.vomag),
                           '{:7.5f}'.format(oref.voang * 180 / np.pi)]

                mainlist.append(sublist)
                rowno.append('Bus ' + str(iloop + 1))
            iloop += 1
        # Present table
        if tpres:
            title = 'Bus Voltages'
            colind = ['Bus no', 'Vmag', 'Theta']
            colw = [0.12, 0.22, 0.22, 0.22, 0.22]
             #Ingrid            self.tableplot(mainlist, title, colind, rowno,
              ↪ columncol=[], rowcol=[], colw=colw)

    # Display the voltages.
    def dispVoltRange(self, fromBus=0, toBus=0, tpres=False, vmin=0.9, vmax=1.1,
     ↪  case=None):
        """
        Desc:    Display voltages at all buses below or equal to the limit vmax
        Input:   tpres= False (Display in tableformat if True)
                 fromBus and toBus defines the block, If tpres=True, it will
↪   display 13 lines from fromBus
                 vmax = Upper voltage limit (default 1.1 pu)
                 vmin = Lower voltage limit (defualt 0.9 pu)
        Returns: None
        """
        mainlist = []
        rowno = []
        if toBus == 0:
            toBus = len(self.BusList)
        #        if tpres:
        #            toBus = np.minimum(fromBus + 13, toBus)

        if fromBus < len(self.BusList): # Check legal range
            iloop = fromBus
        else:
            print(' Bus :', fromBus, ' does not exist')
            return()
```

```
        print(' ')
        while iloop < toBus:
            oref = self.BusList[iloop]
            if vmax >= oref.vomag >= vmin:
                if not tpres:
                    print(' Bus no :', '{:4.0f}'.format(oref.busnum),
                          ' Vmag :', '{:7.5f}'.format(oref.vomag),
                          ' Theta :', '{:7.5f}'.format(oref.voang * 180 / np.pi))
                # Prepare for graphics presentation
#Ingrid            sublist = ['{:4.0f}'.format(oref.busnum),
#Ingrid                       '{:7.5f}'.format(oref.vomag),
#Ingrid                       '{:7.5f}'.format(oref.voang * 180 / np.pi)]
            sublist = ['{:7.5f}'.format(oref.vomag),
                       '{:7.5f}'.format(oref.voang * 180 / np.pi)]
            mainlist.append(sublist)
#Ingrid            rowno.append('Bus ' + str(iloop + 1))
            rowno.append('B' + str(iloop + 1)) #Ingrid
            iloop += 1
        # Present table
        if tpres:
            title = 'Bus Voltages'
#Ingrid            colind = ['Bus nr', 'Vmag', 'Theta']
#Ingrid            colw = [0.12, 0.22, 0.22, 0.22, 0.22]
            colind = ['Vmag', 'Theta'] #Ingrid
             #Ingrid            self.tableplot(mainlist, title, colind, rowno, columncol=[], rowcol=[], colw=colw)
            self.tableplot(mainlist, title, colind, rowno, case)

    # Display voltage estimate for a changes in active or reactive load on a bus
    def dispVoltEst(self, bus=0, deltap=0.0, deltaq=0.0, tpres=False):
        """ The method estimates the voltages for a change in active or reactive load at a bus
        deltap and deltaq must reflect the change (negative by load reduction)
        """
        itr = bus - 1
        mainlist = []
        rowno = []
        iloop = 0
        while self.BusList[itr].toline:
            busobj = self.BusList[itr]
            voltest = busobj.vomag + deltap * (1 + busobj.dPlossdP) * busobj.dVdP + deltaq * (
                    1 + busobj.dQlossdQ) * busobj.dVdQ
            if not tpres:
                print(' Bus no :', '{:4.0f}'.format(busobj.busnum),
                      ' Vmag :', '{:7.4f}'.format(busobj.vomag),
                      ' Vest :', '{:7.4f}'.format(voltest))
            # Prepare for graphics presentation
            if iloop < 14:
                sublist = ['{:4.0f}'.format(busobj.busnum),
                           '{:7.4f}'.format(busobj.vomag),
                           '{:7.4f}'.format(voltest)]
                mainlist.append(sublist)
                rowno.append('Bus ' + str(iloop + 1))
            iloop += 1
            itr = busobj.toline.fbus - 1
```

```
        # Present table
        if tpres:
            title = 'Voltage estimat for changed injection of P and Q'
            colind = ['Bus no', 'Bus volt', 'Volt est']
            colw = [0.12, 0.22, 0.22, 0.22, 0.22]
             #Ingrid            self.tableplot(mainlist, title, colind, rowno, columncol=[], rowcol=[], colw=colw)

    # Display the voltage sensitivities
    def dispVoltSens(self, fromBus=0, toBus=0, tpres=False):
        """
        Desc:    Display Load sensitivities for change in voltage at all buses
        Input:   tpres= False (Display in table format if True)
                 fromBus and toBus defines the block, If tpres=True, it will display 13 lines from fromBus
        Returns: None
        """
        mainlist = []
        rowno = []
        if toBus == 0:
            toBus = len(self.BusList)
        if tpres:
            toBus = np.minimum(fromBus + 13, toBus)

        if fromBus < len(self.BusList): # Check legal range
            iloop = fromBus
        else:
            print(' Bus :', fromBus, ' does not exist')
            return()

        print(' ')
        while iloop < toBus:
            oref = self.BusList[iloop]
            if not tpres:
                print(' Bus no :', '{:4.0f}'.format(oref.busnum),
                      ' dV/dP :', '{:7.5}'.format(oref.dVdP * (1.0 + oref.dPlossdP)),
                      ' dPloss/dP :,{:7.5}'.format(oref.dPlossdP),
                      ' dPloss/dQ :,{:7.5}'.format(oref.dPlossdQ),
                      ' dV/dQ :', '{:7.5}'.format(oref.dVdQ * (1.0 + oref.dPlossdQ)),
                      ' dQloss/dQ :,{:7.5}'.format(oref.dQlossdQ),
                      ' dQloss/dP :,{:7.5}'.format(oref.dQlossdP))

            # Prepare for graphics presentation
            sublist = ['{:4.0f}'.format(oref.busnum),
                       '{:7.5}'.format(oref.dVdP * (1.0 + oref.dPlossdP)),
                       '{:7.5}'.format(oref.dPlossdP),
                       '{:7.5}'.format(oref.dPlossdQ),
                       '{:7.5}'.format(oref.dVdQ * np.sqrt((1.0 + oref.dPlossdQ) ** 2 + oref.dQlossdQ ** 2)),
                       '{:7.5}'.format(oref.dQlossdQ),
                       '{:7.5}'.format(oref.dQlossdP)
                       ]

            mainlist.append(sublist)
            rowno.append('Bus ' + str(iloop + 1))
            iloop += 1
```

```
        # Present table
        if tpres:
            title = 'Bus Voltage sensitivites to changes in load and loss'
            colind = ['Bus no', 'dV/dP', 'dPloss/dP', 'dPloss/dQ', 'dV/dQ',
             ↪  'dQloss/dQ', 'dQloss/dP']
             #Ingrid            self.tableplot(mainlist, title, colind, rowno,
              ↪  columncol=[], rowcol=[])

    # Display loss sensitivities
    def dispLossSens(self, fromBus=0, toBus=0, tpres=False):
        """
        Desc:    Display Loss sensitivities for change in active or reactive
↪   injection at all buses
        Input:   tpres= False (Display in tableformat if True)
                 fromBus and toBus defines the block, If tpres=True, it will
↪   display 13 lines from fromBus
        Returns: None
        """
        mainlist = []
        rowno = []
        if toBus == 0:
            toBus = len(self.BusList)
        if tpres:
            toBus = np.minimum(fromBus + 13, toBus)

        if fromBus < len(self.BusList): # Check legal range
            iloop = fromBus
        else:
            print(' Bus :', fromBus, ' does not exist')
            return()

        print(' ')
        while iloop < toBus:
            oref = self.BusList[iloop]
            if not tpres:
                print(' Bus no :', '{:4.0f}'.format(oref.busnum),
                      ' dV/dP :', '{:7.5}'.format(oref.dVdP * (1.0 +
                       ↪   oref.dPlossdP)),
                      ' dPloss/dP :,{:7.5}'.format(oref.dPlossdP),
                      ' dPloss/dQ :,{:7.5}'.format(oref.dPlossdQ),
                      ' dV/dQ :', '{:7.5}'.format(oref.dVdQ * (1.0 +
                       ↪   oref.dQlossdQ)),
                      ' dP2loss/dP2 :,{:7.5}'.format(oref.dP2lossdP2 - 1.0),
                      ' dP2loss/dQ2 :,{:7.5}'.format(oref.dP2lossdQ2 - 1.0))

            # Prepare for graphics presentation
            sublist = ['{:4.0f}'.format(oref.busnum),
                       '{:7.5}'.format(oref.dVdP * (1.0 + oref.dPlossdP)),
                       '{:7.5}'.format(oref.dPlossdP),
                       '{:7.5}'.format(oref.dPlossdQ),
                       '{:7.5}'.format(oref.dVdQ * (1.0 + oref.dQlossdQ)),
                       '{:7.5}'.format(oref.dP2lossdP2 - 1.0),
                       '{:7.5}'.format(oref.dP2lossdQ2 - 1.0)
                       ]

            mainlist.append(sublist)
            rowno.append('Bus ' + str(iloop + 1))
            iloop += 1
```

```
        # Present table
        if tpres:
            title = 'Bus Voltage sensitivites to changes in load and loss'
            colind = ['Bus no', 'dV/dP', 'dPloss/dP', 'dPloss/dQ', 'dV/dQ',
             ↪  'd2Ploss/dP2', 'd2Ploss/dQ2']
             #Ingrid            self.tableplot(mainlist, title, colind, rowno,
              ↪  columncol=[], rowcol=[])

    # Display loss sensitivities for active power injection
    def dispLossSensP(self, fromBus=0, toBus=0, tpres=False):
        """
        Desc:    Display Loss sensitivities for change in active or reactive
↪   injection at all buses
        Input:   tpres= False (Display in tableformat if True)
                 fromBus and toBus defines the block, If tpres=True, it will
↪   display 13 lines from fromBus
        Returns: None
        """
        mainlist = []
        rowno = []
        if toBus == 0:
            toBus = len(self.BusList)
        if tpres:
            toBus = np.minimum(fromBus + 13, toBus)

        if fromBus < len(self.BusList): # Check legal range
            iloop = fromBus
        else:
            print(' Bus :', fromBus, ' does not exist')
            return()

        print(' ')
        while iloop < toBus:
            oref = self.BusList[iloop]
            if not tpres:
                print(' Bus no :', '{:4.0f}'.format(oref.busnum),
                      ' dV/dP :', '{:7.5}'.format(oref.dVdP * (1.0 +
                       ↪  oref.dPlossdP)),
                      ' dPloss/dP :,{:7.5}'.format(oref.dPlossdP),
                      ' dP2loss/dP2 :,{:7.5}'.format(oref.dP2lossdP2 - 1.0),
                      ' Loss Ratio P :,{:7.5}'.format(oref.lossRatioP))

            # Prepare for graphics presentation
            sublist = ['{:4.0f}'.format(oref.busnum),
                       '{:7.5}'.format(oref.dVdP * (1.0 + oref.dPlossdP)),
                       '{:7.5}'.format(oref.dPlossdP),
                       '{:7.5}'.format(oref.dP2lossdP2 - 1.0),
                       '{:7.5}'.format(oref.lossRatioP)
                       ]

            mainlist.append(sublist)
            rowno.append('Bus ' + str(iloop + 1))
            iloop += 1
        # Present table
        if tpres:
            title = 'Bus Voltage sensitivites to changes in load and loss'
            colind = ['Bus no', 'dV/dP', 'dPloss/dP', 'd2Ploss/dP2',
                      'Loss Ratio P']
```

```
            colw = [0.12, 0.22, 0.22, 0.22, 0.22]
             #Ingrid            self.tableplot(mainlist, title, colind, rowno,
              ↪   columncol=[], rowcol=[], colw=colw)

    # Display loss sensitivities for reactive power injections
    def dispLossSensQ(self, fromBus=0, toBus=0, tpres=False):
        """
        Desc:    Display Loss sensitivities for change in active or reactive
↪   injection at all buses
        Input:   tpres= False (Display in tableformat if True)
                 fromBus and toBus defines the block, If tpres=True, it will
↪   display 13 lines from fromBus
        Returns: None
        """
        mainlist = []
        rowno = []
        if toBus == 0:
            toBus = len(self.BusList)
        if tpres:
            toBus = np.minimum(fromBus + 13, toBus)

        if fromBus < len(self.BusList): # Check legal range
            iloop = fromBus
        else:
            print(' Bus :', fromBus, ' does not exist')
            return()

        print(' ')
        while iloop < toBus:
            oref = self.BusList[iloop]
            if not tpres:
                print(' Bus no :', '{:4.0f}'.format(oref.busnum),
                      ' dV/dQ :', '{:7.5}'.format(oref.dVdQ * (1.0 +
                      ↪   oref.dQlossdQ)),
                      ' dPloss/dQ :,{:7.5}'.format(oref.dPlossdQ),
                      ' dP2loss/dQ2 :,{:7.5}'.format(oref.dP2lossdQ2 - 1.0),  #
                      ↪   1.0 ref value
                      ' Loss Ratio Q :,{:7.5}'.format(oref.lossRatioQ))

            # Prepare for graphics presentation
            sublist = ['{:4.0f}'.format(oref.busnum),
                       '{:7.5}'.format(oref.dVdQ * (1.0 + oref.dQlossdQ)),
                       '{:7.5}'.format(oref.dPlossdQ),
                       '{:7.5}'.format(oref.dP2lossdQ2 - 1.0),
                       '{:7.5}'.format(oref.lossRatioQ)
                       ]

            mainlist.append(sublist)
            rowno.append('Bus ' + str(iloop + 1))
            iloop += 1
        # Present table
        if tpres:
            title = 'Bus Voltage sensitivites to changes in load and loss'
            colind = ['Bus no', 'dV/dQ', 'dPloss/dQ', 'd2Ploss/dQ2',
                      'Loss Ratio Q']
            colw = [0.12, 0.22, 0.22, 0.22, 0.22]
#Ingrid            self.tableplot(mainlist, title, colind, rowno, columncol=[],
↪   rowcol=[], colw=colw)
```

```
    # General table controlled by the application
#Ingrid    def tableplot(self, table_data, title, columns, rows, columncol=None,
↪   rowcol=None, colw=None)
    def tableplot(self, table_data=[], title=None, columns=[], rows=[],
    ↪   case=None, Lists=[]): #Ingrid
        """
        Desc:   Make a table of the provided data. There must be a row and a
↪   column
                data correpsonding to the table
        Input:  table_data  - np.array
                title - string
                columns - string vector
                rows    - string vector
                columncol - colors of each column label (default [])
                rowcol - colors of each row lable
        """

        rowcol=[]
        new_rows=[]
        new_table=[]

        iloop = 0
        limitbreach=0

        colLists = [zero, small, medium, large, over]
        colstandard = [buses, lines, cases]
        if rows[iloop][0]=='B':
            x = 0
        elif rows[iloop][0]=='L':
            x = 1
        else:
            x = 2

        tdim = np.shape(table_data)

        while iloop < tdim[0]:
            c = colstandard[x]

            if len(Lists) > 0:
                #Lists = [zero, small/under, medium, large, over]
                for List in Lists:
                    ind = Lists.index(List)
                    for obj in List:
                        if rows[iloop][0]=='B':
                            if int(rows[iloop][1:])==obj.busnum:
                                c = colLists[ind]
                        if rows[iloop][0]=='L':
                            if int(rows[iloop][1:])==obj.linenum:
                                c = colLists[ind]

            if rows[iloop][0]=='B':
                vomag = float(table_data[iloop][0])
                voang = float(table_data[iloop][1])
                if vomag==1.0 and voang==0.0:
                    c = supply
```

```
            #if c in colLists and case[0:2]!='sp': # or case[:6]=='impact' or
            ↪   case[0:2]=='su':
#            if c in colLists or case[:6]=='impact':
            if case[0:2]=='su':
                limitbreach=1
                rowcol.append(c)
                new_rows.append(rows[iloop])
                new_table.append(table_data[iloop])
            iloop += 1
#        print(rowcol)



        #Tabulate:
        if limitbreach==1:
            para = ''
            fr=''
            w=.2
            pad=0
            columncol = [cases]*len(columns)
            if title=='Bus Voltages':
                para = 'v_'
                fr = rows[0]
            elif title=='Transmission line flow':
                para = 'f_'
                fr = rows[0]
                w=.16
            if case[-1]=='V' or case[-1]=='F':
                columncol = colLists
                if case[-1]=='V':
                    w=.26
            colw = [w]*len(columns)

            fig = plt.figure(dpi=150)
            ax = fig.add_subplot(1, 1, 1)
            ax.axis('off')
            table = ax.table(cellText=new_table, rowLabels=new_rows,
            ↪   colColours=columncol, rowColours=rowcol, colLabels=columns,
            ↪   colWidths=colw, loc='center', cellLoc='center')
            table.set_fontsize(12)
    #Ingrid        table.scale(1,1.5)
    #Ingrid        ax.set_title(title, fontsize=14)

#Ingrid-------------------
            fig.savefig('/Users/ingrid/Documents/stud/prog/fig/' + case + para +
            ↪   fr + ".png", bbox_inches='tight',pad_inches=pad)
            plt.close(fig)
    #Ingrid        plt.show()
#--------------------------

    # Display total losses
#Ingrid    def dispLosses(self):
    def dispLoss(self):
        pline = 0.0
        qline = 0.0
        for x in self.LineList:
            pline += x.ploss
            qline += x.qloss
```

```python
#Ingrid        print('\n', 'Ploss:', pline, '   Qloss:', qline)

        P_Loss = '{:7.4f}'.format(pline)
        Q_Loss = '{:7.4f}'.format(qline)
        return P_Loss, Q_Loss #Ingrid


    # Display total load (no voltage correction)
    def dispLoad(self):
        aload = 0.0
        rload = 0.0
        for x in self.BusList:
            pla, qla, dPdV1, dQdV1 = self.getload(x)
            aload += pla  # Add local loads
            rload += qla
#Ingrid        print('\n', 'Total load  P: ', aload, '   Q: ', rload, '  Losses: P',
#Ingrid              BusList[1].pblossds, '   Q: ', BusList[1].qblossds)
        P_Load = '{:7.4f}'.format(aload)
        Q_Load = '{:7.4f}'.format(rload)
        return P_Load, Q_Load #Ingrid

    def zeroxq(self):
        for a in self.LineList:
            a.x = 0.0
        for a in self.BusList:
            a.qload = 0.0
    # Prepare the case
    def initialize(self, startBus):
        """
        Builds the system, creates the loafd flow object and prepared the additional configuration
        """
        self.flatStart()    # be sure to have a flat start with topology changes
        self.config3()  # Set up additional configuration based on input data
        self.findtree(startBus)     # Identify the tree from the given starting point
     #   if startBus != 1:
        self.config3()      # Update based on the the starting point
        self.topology = self.mainstruct4(startBus = startBus)   # Build the structure

#
# Demo case (Illustration of how to build up a script)
#
# BusList, LineList = BuildSystem3()  # Import data from Excel file
# dlf = DistLoadFlow3(BusList, LineList)  # Create object
# dlf.config3()  # Set up additional configuration
# dlf.findtree(1)
# #svc = DistribObjects3.SVC(dlf.BusList[43], svcstat=1, vref=1, injQmax=1.0, injQmin=0.0, slopeQ=0.05 )
# #dlf.BusList[43].comp = svc
# dlf.config3()
# dlf.topology = dlf.mainstruct4(startBus = 1)

#BusList, LineList = BuildSystem3()  # Import data from Excel file
#dlf = DistLoadFlow3(BusList, LineList)  # Create object
#
```

```
#dlf.initialize(startBus=1) # Initialize
#
#dlf.DistLF(epsilon=0.00001)     # Solve the case
#
#dlf.dispTree(dlf.topology, feeders=[1], LEC= [31, 48], lowVolt = [52, 53, 65, 64], overload=[(62, 63), (63, 64)])
#
# New feeder

#dlf.initialize(startBus=50) # Initialize

#dlf.DistLF(epsilon=0.00001)     # Solve the case

#dlf.dispTree(dlf.topology, feeders=[50], LEC= [31, 48], lowVolt = [52, 53, 65, 64], overload=[(62, 63), (63, 64)])

# dlf.checkOverLoad()
# dlf.checkOverflow()
# dlf.resetBuses()
# dlf.resetBuses()
# dlf.resetBuses()
# dlf.neededInjection(42, 'reactive')
# dlf.neededInjectionLine2(11)
#dlf.dispVolt(fromBus=35, tpres=True)
#dlf.dispLossSens(fromBus=35, tpres=True)
#dlf.dispFlow(fromLine=10, tpres=True)
# dlf.disconnectBus(53)
# dlf.connectLine2(70)
#dlf.findtree()
#dlf.config3()
#dlf.topology = dlf.mainstruct4()  # Set up the configuration for recursive

#Checking for splitting the network
#BusList2, LineList2 = BuildSystem3()
#dlf2 = DistLoadFlow3(BusList2, LineList2)
#dlf2.config3()
#dlf.disconnectBus(10)
#dlf2.disconnectBus(10)
#dlf2.connectLine2(69)
#dlf.findtree(1)
#dlf2.findtree(11)
#dlf.config3()
#svc = DistribObjects3.SVC(dlf.BusList[59], svcstat=1, vref=0.97, injQmax=1.0, injQmin=0.0, slopeQ=0.05 )
#dlf.BusList[44].comp = svc
#dlf2.config3()
#dlf.topology = dlf.mainstruct4()
#dlf2.topology = dlf2.mainstruct4(11)
#print('Topology first network: ')
#dlf.ForwardSearch(dlf.topology)
#print('Topology second network: ')
#dlf2.ForwardSearch(dlf2.topology)
# dlf.ForwardSearch(dlf.topology)
# dlf.UpdateControl(BusList)                    # Update the scaling factors in case of voltage control
#dlf.DistLF(epsilon=0.00001)  # Solve load flow
#dlf.overflow()
#connected = dlf.connectedBuses(dlf.topology)
```

```
#dlf.potential(dlf.BusList[60])
#dlf.highestPotential(connected)
#dlf.findCompensation(dlf.BusList[61])
#dlf.resetBuses()
#dlf.dispVolt(fromBus=0, tpres=True)  # Display voltages for the firste 13 buses
#dlf.dispVolt(fromBus=15, tpres=True)
#dlf.dispVolt(fromBus=38, toBus=46, tpres=True)
#dlf.dispFlow(tpres=True)
#dlf.dispLossSens(fromBus=15, tpres=True)
#dlf.dispLossSens(fromBus=38, toBus=46, tpres=True)
#dlf.dispVoltSens(fromBus=15, tpres=True)
#dlf.dispVoltSens(fromBus=38, toBus=46, tpres=True)
#dlf2.DistLF(epsilon=0.00001)  # Solve load flow
#dlf2.resetBuses()
#dlf2.dispVolt(fromBus=0, tpres=True)  # Display voltages for the firste 13
↪  buses
#dlf2.dispVolt(fromBus=15, tpres=True)
#dlf2.dispVolt(fromBus=50, tpres=True)
#dlf.ForwardSearch(dlf.topology)
# dlf.dispVolt(fromBus=53,toBus=65, tpres=True)
# dlf.dispVoltSens(fromBus=53,toBus=65,tpres=True)
# dlf.dispLossSens(fromBus=53,toBus=65,tpres=True)
##dlf.dispVoltSens(fromBus=10, toBus=23, tpres=True)  # Voltage sensitivities for
↪  reduced load at the same bus and the sensitivity in reduced losses
##dlf.dispLossSens(fromBus=10, toBus=23, tpres=True)  # Loss sensitivities for
↪  reduced load at the same bus and the rate of change of loss sensitivities
#dlf.dispFlow(tpres=True)                      # Display flow on transmission
↪  lines (in graphic pres only 13 is deplayed (spes start point)
##dlf.dispFlow(fromLine=10, tpres=True)
```

Return to the preface, Section 1.2, Section 2.1 or Algorithm B.2's flow chart in Figure 2.2.

D PyDSAL's class objects

See Table 1.2 for an overview of PyDSAL's scripts. Algorithm B.4 is commented on in Section 4..

Algorithm B.4: *`DistribObjects-vIngrid.py`*

```python
#!/usr/bin/python
# Copyright (c) 2021, Olav B. Fosso, NTNU
#
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without modification,
# are permitted provided that the following conditions are met:
#
#     * Redistributions of source code must retain the above copyright notice,
#       this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above copyright notice,
#       this list of conditions and the following disclaimer in the documentation
#       and/or other materials provided with the distribution.
# Definition of common classes

class Bus:
    'Common base class for all distribution buses'
    busCount = 0

    def __init__(self, busnum=0, pload=0.0, qload=0.0, ZIP=[0.0, 0.0 ,1.0], vset=0.0, iloss=0, pqcostRatio=100,vmin=0.9,vmax=1.1, island=0):
        self.busnum = busnum
        self.busext = 0
        self.pload = pload
        self.qload = qload
        self.ZIP = ZIP
        self.vset = vset
        self.iloss = iloss
        self.pqcostRatio = pqcostRatio
        self.vmin = vmin
        self.vmax = vmax
        self.controlScale = 1.0     # Scaling factor to be used during voltage control and loss minimization
        self.comp = 0               # Compensation present
        self.pv = 0                 # PV present
        self.battery = 0            # Battery present
        self.v2g = 0                # V2G present
        self.ploadds = 0.0
        self.qloadds = 0.0
        self.pblossds = 0.0
        self.qblossds = 0.0
        self.dPdV = 0.0
        self.dQdV = 0.0
        self.dVdP = 0.0
        self.dVdQ = 0.0
        self.dPlossdP = 0.0
        self.dPlossdQ = 0.0
        self.dQlossdP = 0.0
```

```
        self.dQlossdQ = 0.0
        self.dP2lossdP2 = 1.0   # To be able to run the voltage optimization also in the first iteration
        self.dP2lossdQ2 = 1.0   # To be able to run the voltage optimization also in the first iteration
        self.lossRatioP = 0.0
        self.lossRatioQ = 0.0
        self.voang = 0.0
        self.vomag = 1.0
        self.busname = 'Bus' + str(busnum)
        self.toline = 0
        self.fromline = 0
        self.tolinelist = []
        self.nextbus = []
        Bus.busCount += 1

class Line:
    'Common base class for all distribution lines'
    lineCount = 0

    def __init__(self, fbus=0, tbus=0, r=0.0, x=0.0, ratea=0.0, ibstat=1, reserve = 0):
        self.fbus = fbus
        self.tbus = tbus
        self.linenum = 0 #Ingrid. To use in AddEdges (dispGraph).
        self.r = r
        self.x = x
        self.ratea = ratea
        self.ibstat = ibstat
        self.ploss = 0.0
        self.qloss = 0.0
        self.reserve = reserve
        Line.lineCount += 1


class Statcom:
    'Common class for Statcom'
    statcomCount = 0
    def __init__(self, bus, scstat = 1, vref=0.0, injQmax = 0.0, injQmin = 0.0, slopeQ = 0.0 ):
        self.bus = bus
        self.scstat = scstat
        self.vref = vref
        self.injQmax = injQmax
        self.injQmin = injQmin
        self.qinj = 0.0
        self.slopeQ = slopeQ
        Statcom.statcomCount += 1


class SVC:
    'Common class for Static Var Compensator'
    svcCount = 0
    def __init__(self, bus, cmode = 0, svcstat = 1, vref=0.0, injQmax = 0.0, injQmin = 0.0, slopeQ = 0.0 ):
        self.bus = bus
        self.cmode = cmode #La til denne.
        self.stat = svcstat
```

```
        self.vref = vref
        self.vprev = vref
        self.injQmax = injQmax
        self.injQmin = injQmin
        self.qinj = 0.0
        self.slopeQ = slopeQ
        SVC.svcCount += 1

class Battery:
    'Common class for Batteries'
    batteryCount = 0
    def __init__(self, bus, cmode = 0, svcstat = 1, vref=0.0, injPmax = 0.0,
    ↪   injPmin = 0.0, injQmax = 0.0, injQmin = 0.0, slopeP = 0.0, slopeQ = 0.0
    ↪   ):
        self.bus = bus
        self.cmode = cmode #La til denne.
        self.stat = svcstat
        self.vref = vref
        self.vprev = vref
        self.injPmax = injPmax
        self.injPmin = injPmin
        self.injQmax = injQmax
        self.injQmin = injQmin
        self.pinj = 0.0
        self.qinj = 0.0
        self.Estorage = 0.0
        self.slopeP = slopeP
        self.slopeQ = slopeQ
        Battery.batteryCount += 1

class V2G:
    'Common class for Electrical Vehicles'
    v2gCount = 0
    def __init__(self, bus, cmode = 0, v2gstat = 1, vref=0.0, injPmax = 0.0,
    ↪   injPmin = 0.0, injQmax = 0.0, injQmin = 0.0, slopeP = 0.0, slopeQ = 0.0
    ↪   ):
        self.bus = bus
        self.cmode = cmode #La til denne.
        self.stat = v2gstat
        self.vref = vref
        self.vprev = vref
        self.injPmax = injPmax
        self.injPmin = injPmin
        self.injQmax = injQmax
        self.injQmin = injQmin
        self.pinj = 0.0
        self.qinj = 0.0
        self.Estorage = 0.0
        self.slopeP = slopeP
        self.slopeQ = slopeQ
        V2G.v2gCount += 1


class Capacitor:
    'Common class for capacitors'
    capacitorCount = 0
    def __init__(self, bus, capstat = 1, vref=0.0, blockSize = 0.0, numBlocks =
    ↪   1):
```

```
        self.bus = bus
        self.capstat = capstat
        self.vref = vref
        self.blockSize = blockSize
        self.numBlocks = numBlocks
        self.currentStep = 0
        Capacitor.capacitorCount += 1

class PV:
    'Common class for PhotoVoltaic (PV)'
    pvCount = 0
    def __init__(self, bus, pvstat = 1, cmode = 1, vref=0.0, convCap = 0.0, injPmax = 0.0, injPmin = 0.0, injQmax = 0.0, injQmin = 0.0, slopeP = 0.0, slopeQ = 0.0 ):
        self.bus = bus
        self.stat = pvstat
        self.cmode = cmode          # cmode = 1 (PV) , cmode = 2 (P - droop Q, cmode = 3 (Droop P, droop Q)
        self.vref = vref            # Voltage reference - interpreted according to the control mode (cmode)
        self.vprev = vref           # Needed for droop control - iterative procedure
        self.convCap = convCap      # Total converter capability - S^2 = P^2 + Q^2 - limits calculated accordingly or specified
        self.injPmax = injPmax
        self.injPmin = injPmin
        self.injQmax = injQmax
        self.injQmin = injQmin
        self.pinj = 0.0
        self.qinj = 0.0
        self.slopeP = slopeP
        self.slopeQ = slopeQ
        PV.pvCount += 1
```

E PyDSAL's selector of spreadsheets

See Table 1.2 for an overview of PyDSAL's scripts.

Algorithm B.5: `MenuFunctions-v2.py`

```
# Copyright (c) 2021, Olav B. Fosso, NTNU
#
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
↪   modification,
# are permitted provided that the following conditions are met:
#
#     * Redistributions of source code must retain the above copyright notice,
#       this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above copyright
↪   notice,
#       this list of conditions and the following disclaimer in the
↪   documentation
#       and/or other materials provided with the distribution.

from tkinter import *

from tkinter.colorchooser import askcolor
from tkinter.filedialog   import askopenfilename


def GetFileName(filext="*"):
    """ Returns a file name - a file has to be chosen
        Input: filext = "*" - File extention in the first list
    """
    root = Tk()
    file = ""
    if filext == "*":
        fileclass = "All Files"
        fitype = "*.*"
    if filext == "xls":
        fileclass = 'Excel File'
        fitype = "*." + filext
    while file == "":
        file = askopenfilename(filetypes=((fileclass,fitype),
                                          ("Text File","*.txt*"),
                                          ("LP Files","*.lp")),
                               title= "Choose a file")
        root.withdraw()
        print(file)
    return file


def ViewFileName(filext="*"):
    """ View files - Choose one or cancel
        Input: filext = "*" - File extention in the first list
    """
    root = Tk()
    file = ""
    if filext == "*":
```

```
        fileclass = "All Files"
        fitype = "*.*"
    elif filext == "xls":
        fileclass = "Excel File"
        fitype = "*." + filext
    file = askopenfilename(filetypes=((fileclass,fitype),
                                          ("Text File","*.txt*"),
                                          ("LP Files","*.lp"),
                                      ("Excel Files","*.xls")),
                               title= "Choose a file")
    root.withdraw()
    print(file)
    return file

#file = ViewFileName()

def OpenFile():
#    from tkinter import filedialog
#    from tkinter import *

    root = Tk()
    root.filename =  filedialog.askopenfilename(title = "Select file",filetypes = (("Excel Files","*.xls"),("all files","*.*")))
    return root.filename
```

F PyDSAL's reader of selected spreadsheet

See Table 1.2 for an overview of PyDSAL's scripts.

Algorithm B.6: `BuildSystem-vIngrid.py`

```
#!/usr/bin/python
# Copyright (c) 2021, Olav B. Fosso, NTNU
#
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
↪  modification,
# are permitted provided that the following conditions are met:
#
#     * Redistributions of source code must retain the above copyright notice,
#       this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above copyright
↪  notice,
#       this list of conditions and the following disclaimer in the
↪  documentation
#       and/or other materials provided with the distribution.

import numpy as np
from DistribObjects_vIngrid import *
import pandas as pd

#from MenuFunctions_v2 import ViewFileName #Jeg endret dette, for å slippe å
↪  velge xcel-filen hele veien.

def BuildSystem3():
    def renumber(BusList, LineList):
        iloop1 = 0
        sbase = 1   # assume that input values of load are in PU.
        temp = np.zeros(2000,dtype=int)
        while iloop1 < len(BusList):
            obj = BusList[iloop1]
            obj.busext = obj.busnum
            obj.busnum = iloop1 +1
            temp[obj.busext] = obj.busnum
            obj.pload = obj.pload/sbase
            obj.qload = obj.qload / sbase
            iloop1 += 1

        iloop1 = 0
        while iloop1 < len(LineList):
            obj = LineList[iloop1]
            obj.fbus = temp[obj.fbus]
            obj.tbus = temp[obj.tbus]
            iloop1 += 1
        return


    BusList = []
    LineList = []
#    file = ViewFileName(filext="xls") #Ingrid
    file = "Cineldi124BusPyDSAL_Load_65.xls" #Ingrid
```

```
    xls = pd.ExcelFile(file)
    df2 = pd.read_excel(xls, 'Bus')
    values = df2.values
    # Read Bus data  --------------------------------------------
    iloop = 0
    # print(' ')
    while iloop < len(values):
        BusList.append(Bus(busnum=int(values[iloop, 0]), pload=values[iloop, 2],
         ↪  qload=values[iloop, 3],
                           vmax=values[iloop, 7], vmin=values[iloop, 8]))
        iloop += 1
    df2 = pd.read_excel(xls, 'Branch')
    values = df2.values
    # Read Bus data  --------------------------------------------
    iloop = 0
    # print(' ')
    while iloop < len(values):
        LineList.append(Line(fbus=int(values[iloop, 0]), tbus=int(values[iloop,
         ↪  1]), r=values[iloop, 2],
                             x=values[iloop, 3], ratea=values[iloop, 5],
                             ↪  ibstat=int(values[iloop, 10]),
                             reserve=int(values[iloop, 11])))
        iloop += 1

    renumber(BusList, LineList)

    return BusList, LineList
```

NTNU

Norwegian University of
Science and Technology