

Green Energy and Technology



Francisco Gonzalez-Longatt
José Luis Rueda Torres *Editors*

Advanced Smart Grid Functionalities Based on PowerFactory

EXTRAS ONLINE



Springer

Green Energy and Technology

More information about this series at <http://www.springer.com/series/8059>

Francisco Gonzalez-Longatt
José Luis Rueda Torres
Editors

Advanced Smart Grid Functionalities Based on PowerFactory



Springer

Editors

Francisco Gonzalez-Longatt
Loughborough
UK

José Luis Rueda Torres
Department of Electrical Sustainable Energy
Delft University of Technology
Delft, Zuid-Holland
The Netherlands

ISSN 1865-3529

Green Energy and Technology

ISBN 978-3-319-50531-2

<https://doi.org/10.1007/978-3-319-50532-9>

ISSN 1865-3537 (electronic)

ISBN 978-3-319-50532-9 (eBook)

Library of Congress Control Number: 2017958729

© Springer International Publishing AG 2018, corrected publication 2018

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG part of Springer Nature

The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

Since the publication of the first book on *PowerFactory* applications in power systems in 2014, a number of aspects related to the increased integration of renewables have become even more relevant. The further increase in inverter-based infeed means a subsequent rise in the need to manage its challenging aspects and phenomena of interaction, control and stability. The definition of appropriate grid code requirements to ensure stable and reliable power system operation, considering an increased number of players, market rules and uncertainties, as well as the drastically increasing complexity of system operation tasks, demands sophisticated and flexible software tools that are highly applicable to the engineering requirements.

Due to rapid developments in power systems, specific requirements of advanced software functions and features of the future are difficult to predict and implement, especially considering software design, quality-based implementation and testing, as well as backwards compatibility. The design and development of *PowerFactory* software have always focused on functional integration, modelling precision, strong numeric algorithms and in particular, overall flexibility featuring state-of-the-art applications. To this end, substantial input and requirement definitions are received via the *PowerFactory* user based on the utility sector and industry and from universities and research institutes.

The level of flexibility provided by *PowerFactory* is demonstrated in this new application book on advanced smart grid functions, covering a wide range of highly relevant topics. The various contributions deliver creative engineering solutions and showcase the broad range of applications that can be covered on a sophisticated technical level. Such contributions also encourage continuing confident utilisation of *PowerFactory* by a large and growing user community to tackle today's power engineering tasks. In addition, this book provides further motivation to the *PowerFactory* development team to incorporate advanced functionality and optimise software efficiency and performance.

We thank all authors who have contributed to this new book, initiated by Francisco Gonzalez-Longat and José Luis Rueda Torres, for sharing their valuable *PowerFactory* experiences and solutions. We are confident that these contributions will greatly help to improve and develop further modern and sustainable power systems.

Tübingen, Germany
October 2017

Dr.-Ing. Martin Schmieg
Chairman of Advisory Board DIgSILENT GmbH

*The original version of the book was revised: ESM files have been included.
The erratum to this book is available at
<https://doi.org/10.1007/978-3-319-50532-15>*

Contents

1	Introduction to Smart Grid Functionalities	1
	Francisco Gonzalez-Longatt and José Luis Rueda Torres	
2	Python Scripting for DIgSILENT PowerFactory: Leveraging the Python API for Scenario Manipulation and Analysis of Large Datasets	19
	Claudio David López and José Luis Rueda Torres	
3	Smart Network Planning—Pareto Optimal Phase Balancing for LV Networks via Monte-Carlo Simulations	49
	Benoît Bleitterie, Roman Bolgaryn and Serdar Kadam	
4	Co-simulation with DIgSILENT PowerFactory and MATLAB: Optimal Integration of Plug-in Electric Vehicles in Distribution Networks	67
	J. Garcia-Villalobos, I. Zamora, M. Marinelli, P. Eguia and J. I. San Martin	
5	Probabilistic Load-Flow Using Analysis Using DPL Scripting Language	93
	Francisco Gonzalez-Longatt, S. Alhejaj, A. Marano-Marcolini and José Luis Rueda Torres	
6	Dynamic Stability Improvement of Islanded Power Plant by Smart Power Management System: Implementation of PMS Logic	125
	Hamid Khoshkho and Ali Parizad	
7	Determining Wide-Area Signals and Locations of Regulating Devices to Damp Inter-Area Oscillations Through Eigenvalue Sensitivity Analysis Using DIgSILENT Programming Language	153
	Horacio Silva-Saravia, Yajun Wang and Héctor Pulgar-Painemal	

8	Dynamic Stability Improvement of Islanded Power Plant by Smart Power Management System—Principles, Descriptions and Scenarios	181
	Ali Parizad and Hamid Khoshkho	
9	Wide-Area Measurement, Monitoring and Control: PMU-Based Distributed Wide-Area Damping Control Design Based on Heuristic Optimisation Using DIgSILENT PowerFactory	211
	Amin Mohammadpour Shotorbani, Sajad Madadi and Behnam Mohammadi-Ivatloo	
10	Optimal PMU Placement Framework Under Observability Redundancy and Contingency—An Evolutionary Algorithm Using DIgSILENT Programming Language Module	241
	Mohsen Zare, Rasoul Azizipanah-Abarghooee, Mostafa Malekpour and Vladimir Terzija	
11	Implementation of Slow Coherency Based Controlled Islanding Using DIgSILENT PowerFactory and MATLAB	279
	I. Tyuryukanov, M. Naglič, M. Popov and M. A. M. M. van der Meijden	
12	Peer-to-Peer (P2P) MATLAB–PowerFactory Communication: Optimal Placement and Setting of Power System Stabilizer	301
	Andrei Stativă and Francisco Gonzalez-Longatt	
13	Implementation of the Single Machine Equivalent (SIME) Method for Transient Stability Assessment in DIgSILENT PowerFactory	319
	Jaime Cepeda, Paúl Salazar, Diego Echeverría and Hugo Arcos	
14	Generic DSL-Based Modeling and Control of Wind Turbine Type 4 for EMT Simulations in DIgSILENT PowerFactory	355
	Abdul W. Korai, Elyas Rakhshani, José Luis Rueda Torres and István Erlich	
	Erratum to: Advanced Smart Grid Functionalities Based on PowerFactory	E1
	Francisco Gonzalez-Longatt and José Luis Rueda Torres	

Introduction

Looking beyond 2050, the challenges for reliable and secure operation and planning of sustainable electricity networks will dramatically increase. The drivers of those challenges include the combined effects of transnational grids and higher market pressures, the transformation of generation technologies to meet environmental targets and changes in anticipated future use of electricity. The major change to the way we supply and use energy; building a smarter grid lie at the heart of these changes. In particular, the ability to accommodate significant volumes of decentralised and highly variable renewable generation requires that the network infrastructure must be upgraded to enable smart operation. The reliable and sophisticated solutions to the foreseen issues of the future networks are creating dynamically intelligent application/solutions to be deployed during the incremental process of building the smarter grid. Also, the boundaries between transmission and distribution, which have been fundamental to the way the power industry and its engineering support societies have been organised, will become vague and ultimately disappear. For all the reasons above, there is a clear need to rethink the way we actually operate the power systems in order to meet the economic, technical and security requirements of future smart grids.

Classical approaches to power system planning such as “predict and provide” (the mid-1980s) and “react and provide” (the mid-1990s) are not good enough to face the most basic challenges created by smarter grids. Advanced smart grid functionalities will be deployed in order to increase efficiency, safety, reliability and quality of the future energy networks, transforming the current electricity grids into a fully interactive (customers/operators) service network, developing bulk and dispersed energy storage options and removing obstacles to the massive-scale deployment and effective integration of distributed and renewable energy sources.

The smart grid needs more powerful computing platforms (centralised and dispersed) to handle large-scale data analytic tasks and supports complicated real-time applications.

Sophisticated simulation packages are required by the smarter grids, and new functionalities are required from them: capability to collect data from smart metres and sensors in near real time; integrate with live weather data, geographical

location; perform a predictive analysis on the aggregated data; and provide decision support for integrating storage, renewable energy sources and transportation systems. DIgSILENT has set standards and trends in power system modelling, analysis and simulation for more than 25 years; and PowerFactory 2017 offers major features required by advanced smart grid functionalities: calculation functions, extensions to the data model and data management system.

Scientists who research and teach electricity power systems and smart grids, professionals at electric utilities, consultancy companies, etc., all of them require taking advantage of the most advanced capabilities provided by the power system simulation packages, DIgSILENT PowerFactory.

Also, there is a lack of knowledge on practical/theoretical principles of advanced smart grid functionalities, especially on regarding “how to” apply modern power systems software on their implementation.

This book consolidates some of the most promising advanced smart grid functionalities and a comprehensive set of guidelines for its implementation/evaluation using DIgSILENT PowerFactory.

The book covers the most important aspects of advanced smart grid functionalities, including special aspects of modelling as well as simulation and analysis, e.g. wide-area monitoring, visualisation and control. Key advanced features of modelling and automation of calculations using PowerFactory are presented, e.g. use of DIgSILENT Simulation Language (DSL) and DIgSILENT Programming Language (DPL) for design and simulation of wide-area/smart grid/intelligent control schemes, use of PowerFactory for model identification and dynamic equivalencing and use of an interface with third-party software for solving problems of optimisation in operation/planning of smart grids.

Besides, realistic examples are specially designed to illustrate the application of PowerFactory on the analysis of the specific phenomenon. Step-by-step procedure is used to explain “How to” employ PowerFactory, and concise theoretical discussions are used to empathise physical understanding of the phenomenon. One important contribution of this book is to make publically available in a website all projects, models and script developed in this book. Those files will allow the reader to follow step by step the examples and be capable of reproducing results. Models and scripts developed in this book can be adapted and modified by the reader to extend its use to other cases and problems.

Chapter 1 is dedicated to present an introduction about the DIgSILENT PowerFactory, the most important aspects of advanced smart grid functionalities, including special aspects of modelling as well as simulation and analysis, e.g. wide-area monitoring, visualisation and control; dynamic capability rating, real-time load measurement and management, interfaces and co-simulation for modelling and simulation of hybrid systems. Chapter 2 illustrates the synergic relationship that can be established between DIgSILENT PowerFactory and a set of Python libraries for data analysis using the Python API, and the examples of static and dynamic simulations using the Python API and PowerFactory are presented.

Chapter 3 presents the development of a user-defined tool to minimise the voltage unbalance caused by unsymmetrical loads and generators. The tool

provides the user with a set of switching actions to improve the power distribution over the three phases.

Chapter 4 presents a co-simulation framework developed to test optimal control methods for root-mean-square (RMS) simulations on DIgSILENT PowerFactory. As an example, the implementation of a smart charging control for plug-in electric vehicles in electric distribution networks is explained. The co-simulation framework used digexfun interface, allowing DIgSILENT PowerFactory to send and receive data from other mathematical software APIs such as MATLAB.

Chapter 5 is dedicated to present the development of a DIgSILENT PowerFactory script language (DPL) implementation of a DPL script to perform probabilistic power flow (PLF) using Monte Carlo simulations (MCS) to consider the variability of the stochastic variables in the power system during the assessment of the steady-state performance.

Chapter 6 shows the capability of DIgSILENT PowerFactory to simulate smart grid functionalities; DIgSILENT Programming Language (DPL) is used to model Power Management System (PMS) Logic in automatically detecting of islanding condition as well as executing load/generation shedding in an islanded system to prevent instability. Indeed, this modelling gives the possibility to check the impact of considered PMS logic under different operating conditions on the stability of the system.

Chapter 7 introduces the concept of eigenvalue sensitivity to analyse the installation location and feedback signals of damping regulating devices using DIgSILENT Programming Language. A state-space representation of the linearised system is estimated by dynamic simulations and used to provide two indices based on mode controllability and mode observability.

Chapter 8 presents the main details of a PMS configuration, and its major functions are explained. The impact of the PMS on system stability is analysed through detailed dynamic simulations in DIgSILENT PowerFactory. The chapter presents the details of the DIgSILENT Simulation Language (DSL) modelling of the PMS and the models of turbine governor, excitation system and signals.

Chapter 9 presents the design of a wide-area damping control (WADC) using a power system stabiliser (PSS) and remote PMU data from the wide-area measurement system (WAMS).

Chapter 10 is focused on the demonstration of capabilities of DIgSILENT PowerFactory software for solving the problem of optimal PMU placement in power networks. The optimal placement has been viewed from the perspective of satisfying the observability requirement of power system state estimator. Optimal placement of PMU is formulated as a practical design task, considering some technical challenges like complete network observability, enough redundancy and the concept of zero injection buses under PMU and tie-line critical contingencies. Furthermore, the meta-heuristic techniques by evolutionary computations are programmed as an optimisation toolbox in DIgSILENT Programming Language (DPL). A distinctive characteristic of the presented module is that the evolutionary algorithm is only coded in DPL without using the time-consuming process of

interlinking DIgSILENT PowerFactory with another software package like MATLAB.

Chapter 11 illustrates a basic intentional controlled islanding (ICI) algorithm implemented in PowerFactory. It utilises the slow coherency theory and constrained graph partitioning to promote transient stability and create islands with a reasonable power balance. The algorithm is also capable of excluding specified network branches from the search space. The implementation is based on the coupling of Python and MATLAB programme codes.

Chapter 12 presents the peer-to-peer MATLAB–PowerFactory communication. The method is extremely simple file sharing approach to couple MATLAB and PowerFactory, and it is used to solve an optimisation problem. An illustrative two-area power system is modelled using PowerFactory, and an optimisation algorithm is implemented in MATLAB.

Chapter 13 addresses key aspects concerning the implementation of Single Machine Equivalent (SIME) by using DIgSILENT Programming Language (DPL). An exemplary application on a well-known benchmark power system is then presented and discussed to highlight the feasibility and effectiveness of the implementation in DIgSILENT PowerFactory environment.

In Chap. 14, a new wind turbine (WT) as well as a VSC–HVDC control concept is presented, which determines the converter reference voltage directly without the need of an underlying current controller. Additionally, alternative options for frequency support by the HVDC terminals that can be incorporated into the active power control channel are presented. The implementation steps performed by using DSL programming are presented for the case of EMT simulations.

We would like to thank all authors and invited reviewers of the individual chapters for their continuous and valuable support in the different stages of the preparation of this book.

We make the maximum effort to make the book a useful source of information on the use of PowerFactory and, at the same time, provide the basis for discussion among readers and users with diverse expertise and backgrounds.

Francisco Gonzalez-Longatt
José Luis Rueda Torres

Chapter 1

Introduction to Smart Grid Functionalities

Francisco Gonzalez-Longatt and José Luis Rueda Torres

Abstract Future power system has several challenges. One of them is the major changes to the way of supply and use energy; building a smarter grid lies at the heart of these changes. The ability to accommodate significant volumes of decentralized and highly variable renewable generation requires that the network infrastructure must be upgraded to enable smart operation. The reliable and sophisticated solutions to the foreseen issues of the future networks are creating dynamically intelligent application/solutions to be deployed during the incremental process of building the smarter grid. The smart grid needs more powerful computing platforms (centralized and dispersed) to handle large-scale data analytic tasks and supports complicated real-time applications. The implementation of highly realistic real-time, massive, online, multi-time frame simulations is required. The objective of this chapter is to present a general introduction to the DIgSILENT PowerFactory, the most important aspects of advanced smart grid functionalities, including special aspects of modelling as well as simulation and analysis, e.g. wide area monitoring, visualization, and control; dynamic capability rating, real-time load measurement and management, interfaces and co-simulation for modelling and simulation of hybrid systems. The chapter presents a very well-documented smart grid functionality, and limited cases are explained: Virtual Control Commissioning: connection to SCADA system via OPC protocol, direct connection to Modbus TCP devices, GIS integration with PowerFactory using API. The explained cases allow showing the full potential of PowerFactory connectivity to fulfil the growing requirements of the smart grids planning and operation.

F. Gonzalez-Longatt (✉)

School of Electronic, Electrical and Systems Engineering,
Loughborough University, Loughborough LE11 3TU, UK
e-mail: fglongatt@fglongatt.org; s.m.alhejaj@lboro.ac.uk

J. L. Rueda Torres

Department of Electrical Sustainable Energy, Delft University
of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: J.L.RuedaTorres@tudelft.nl

Keywords API · DIgSILENT · DPL · DSL · Modbus TCP · Modelling
OPC · Simulations · Smart grid

1.1 Introduction

The phrase “smart grid” is, without any doubt, one of the most utilized expressions in the power utilities in recent years. However, the phrase has associated several aspects of the power system that deserves to be clarified and explained. There is not a universal definition of the smart grid, but there are a few coincident characteristics in the definitions.

Initially, the idea of “smart grid” is not new. In fact, the term smart grid has been in use since, at least 2005, when it appeared in the article “*Toward A Smart Grid*” by Massoud Amin and Wollenberg [1].

Looking to the linguistic phrase or expression smart grid, it represents a single noun and two words are used to provide the idiomatic meaning: “smart” and “grid.”

The noun grid is a folkloric or commonly used term, used for more than a century, to address in simplistic way the power system structure. The etymology of the word is clear and appropriately used. On the other hand, the word smart is the relatively new application of the power system (and then indiscriminately used almost everywhere). The Oxford Dictionary [2] defines the word smart in several contexts. However, it represented an adjective in the phrase smart grid and defined as: “...*Having or showing a quick-witted intelligence...*”. The attributive adjective or adjective of quality “smart” in the phrase smart grid is used to indicate the intelligence as an attribute of the power systems.

Now, thinking about the relatively new use of the qualitative adjective smart to describe the modern and future power systems, it looks extreme unfair. The lack of smart attribute is described as dull, stupid, unintelligent, etc.

If you think about the implications behind the absence of the adjective smart in the classical power system, it is clear the negative effects and disrespects on the roots and the past history of the electric power industry and its pioneers.

The power system has used most of the intelligence available to the engineers at the time, and the power systems have been smart since the very beginning. If you look at the synchronous generators, they have been using controllers (governors, automatic voltage regulators, etc.) since the beginning of the power utility history, and the control objectives are still the same as used by the old mechanical systems. Electromechanical protection, first-generation oldest relaying system, has have been in use for many years, and still in use in so many places around the world. Considering the basic principles of mechanics for the applications above requires a vast amount of ingenuity and creativity, it makes evident, the power systems have been intelligent for decades.

The developments in technology produce a natural evolution in the power system. The recent technological developments and integration of them in the

modern power system have enabled adding more advanced features to the power system, adding intelligence.

Now, it is evident that the phrase smart grid is not the best to describe the changes in the power system evolution. The appropriate way to describe the modern and future power system with increased capabilities provided by the technologies should be adding a suffix to the adjective smart. Adding the -er to the end of the word smart, a comparative adjective form is created: smart-er = more than smart.

The idea of a *smart-er grid* coincides with the original vision provided by Amin and Wollenberg [1]: "...To add intelligence to an electric power transmission system." However, it is extremely complex to fully define the concept of intelligence and the mechanism of adding it to the power systems.

The smart grid looks more like a marketing term, rather than a fully technical definition. Consequently, there is no universal and commonly accepted scope of what "smart" is and what it is not. The general consensus is the smart grid should modernize the electric grid to cope with the emergent challenges of the future power system including the new technologies.

The modernization of the power system and fast transition to the smart grid requires several functionalities, and there is not common agreement on them, but the most accepted functionalities include: (i) self-healing from power disturbance events, (ii) enabling active consumers' participation and operating resiliently against attack, (iii) providing power quality and optimizing assets, and (iv) accommodating all generation and enabling new products, services, and markets. More and more functionalities can be found in the literature [3, 4] but a discussion of all of them is beyond the scope of this chapter.

The smart grid functionalities require implementation of devices/services with high computing performance into the grids and the development of a fast communication network between these devices.

A very limited list of smart grid functions includes: fault current limiting; wide area monitoring, visualization, and control; dynamic capability rating; active and flexible dynamic power flow control, adaptive self-healing protections; automatic feeder and line switching (including optimizing performance or restoration), automating islanding operation and reconnection; diagnosis and notification of equipment condition, customer electricity use optimization.

The appropriate development of the smart grid functionalities creates challenges in several sectors of the power system industry, but also in what the hardware and software industry will develop in the future years.

The specific case of the power system analysis software that the power utilities currently use should evolve to cope with the fast-developing smart grid functionalities. Planners and operators are especially interested in power system analysis tool that allows them coping with the new challenges involved with the smart grid functionalities.

Numerous commercial and open-source tools are available for power system analysis (see a detailed list at [5]), each of them has advantages and disadvantages.

- (a) **Free or open-source tools (non-commercial software).** There are a very large number of freely available software tools for power system analysis, and some of them have been created using the MATLAB [6] platform. However, MATLAB-based programs have an important drawback; they require MATLAB software to make use of these software tools; as a consequence, those programs are not totally free! On the other hand, there are some other power system analysis software developed using Python [7]. The use of open-source software (OSS) has allowed the researcher to easily get access to powerful power system analysis routines which allows the possibility of a flexible platform for development. A limited list of non-commercial power system analysis software is included: Dome (Python-based) [8], GridCal, GridLAB-D [8], MatDyn [9], MATPOWER [10], OpenDSS (distribution system simulator) [11], Power Systems Analysis Toolbox (PSAT) [12].
- (b) **Commercial tools.** Power system analysis software has been available in the market for decades. Traditionally, the commercial power system tools have been closed; it means limited access to change solution methods, models, database, etc. However, the software industry has progressively evolved to provide more and more flexibility and access to the user. Today, the majority of the power system analysis software allows the users alternative was to access the software, user-defined models, etc. The use of modern interfaces between software and hardware makes the power system analysis software more and more flexible. However, the main drawback of the commercial tools is still the same, the cost. Many power system analyses are available in the market: ASPEN [13], BCP Switzerland (NEPLAN) [13], CYME [14], ETAP [15], IPSA Power [16], Power Analytics (EDSA) [17], Siemens PTI (PSS/E and SINCAL) [18], DIgSILENT PowerFactory [8].

An important aspect of the development of smart grids is the interaction between multiple grid components. However, the number of active grid components (i.e. components that actively affect the state of the power network by local or centralized algorithms) is increasing in the power system; as a consequence, the complexity of the smart grid is increased [19]. The usage of power system simulations is recognized as very well-established and important method; it has been used for decades for the performance assessment of the power systems.

DIgSILENT is one of the most used power system analysis tools. It is used for modelling, analysis, and simulation of the power system for more than 25 years. DIgSILENT offers the most economical solution, as data handling, modelling capabilities, and overall functionality replace a set of other power system analysis software.

The scientists who research and teach electricity power systems and smart grids, professionals at electric utilities, consultancy companies, etc., all of them require taking advantage of the most advanced capabilities provided by the power system simulation packages DIgSILENT PowerFactory.

This chapter presents a general introduction to the DIgSILENT PowerFactory the most important aspects of advanced smart grid functionalities, including special

aspects of modelling as well as simulation and analysis, e.g. wide area monitoring, visualization, and control; dynamic capability rating, real-time load measurement and management, interfaces and co-simulation for modelling and simulation of hybrid systems.

1.2 Interfaces Provided by DIgSILENT PowerFactory

DIgSILENT offers the most economical solution, as data handling, modelling capabilities, and overall functionality replace a set of other power system analysis software.

Among other functionality, DIgSILENT PowerFactory is capable of calculating power flow, short circuit, harmonic, stability simulations like transients as well as steady state simulations for balanced and unbalanced systems. Beyond the described classical functionalities of power system analysis, DIgSILENT PowerFactory offers a whole set of power system analysis functions dedicated to smart grids planning and operation, electric vehicles, renewable energies and distributed generation integration studies: load-flow with many optimization functions; optimal power flow (OPF); distribution network optimization; open tie point placement; time sweep; state estimation, and much more. But DIgSILENT PowerFactory also offers flexibility in modelling, simulation, and communication.

Interfaces for co-simulation: The extensive integration of power system and ICT infrastructure mandates that the two systems must be studied as a single distributed cyber-physical system [20]. DIgSILENT PowerFactory offers several alternatives to be coupled with other models; (see Fig. 1.1, more detailed explanation can be found in [21]):

- (i) MATLAB: PowerFactory built-in interface (DIgSILENT Simulation Language, DSL) for co-simulation.
- (ii) Dynamic-link library (or DLL): The integration of an external *event-driven* C/C++ DLL (*digexdyn.dll*) is possible. It can be integrated via DSL blocks () (means only available in RMS simulation) by calling the externally defined function (e.g. TCP/IP sockets). The result of the evaluated code is injected back into the RMS simulation by emitting events.
- (iii) OLE for Process Control (OPC): The industrial standard interface—OPC client—is an asynchronous communication and data exchange mechanism used in process interaction. The data transfer is synced automatically in RMS simulation according to the update frequency, but which is only practicable in combination with the real-time PC-clock synchronization. The OPC client can be used with multi-agent systems and controller hardware-in-the-loop. (OLE: Object Linking and Embedding).
- (iv) Remote Procedure Call (RPC): The RPC is a powerful technique for constructing distributed client–server-based applications. The command server functionality is used for *Remote Procedure Calls* (RPCs) with client/server

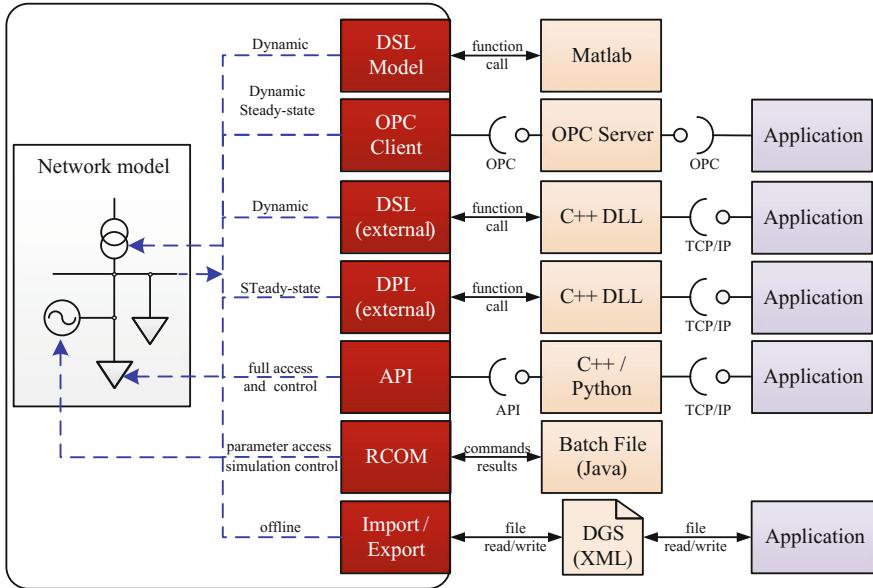


Fig. 1.1 General overview of the interfaces provided by DIgSILENT PowerFactory

communication [21]. Remote communication allows calling the PowerFactory in engine mode¹ (e.g. automated simulation).

- (v) Application programming interface (API): The API encapsulates the internal data model and various simulation functionalities in PowerFactory [21]. The API enables a direct control of PowerFactory provided access to advanced functionality (e.g. co-simulation).
- (vi) DGS: PowerFactory supports a wide set of interfaces. Depending on the specific data exchange task, the user may select the appropriate interface. The DGS (DIgSILENT) is PowerFactory's standard bidirectional interface specifically designed for bulk data exchange with other applications such as GIS and SCADA and, for example, for exporting calculation results to produce Crystal Reports, or to interchange data with any other software package. The DGS allows the exchanging data models and geographical information.

Scripting and automation: DIgSILENT PowerFactory offers two programming languages for scripting: DIgSILENT Programming Language (DPL) and Python programming language [7]. The DPL serves the purpose of offering an interface for automating tasks in the PowerFactory program. The DPL method distinguishes itself from the command batch method in several aspects: it offers decision and flow commands, it allows the definition and use of user-defined variables, DPL has a

¹Details of engine mode operation are found at the DIgSILENT PowerFactory User's Manual.

flexible interface for input–output and for accessing objects, and DPL offers mathematical expressions. Additionally, to DPL it is also possible to use the Python language to write scripts to be executed in PowerFactory. Python language is a very powerful tool, which is integrated into the standard PowerFactory application. It is commonly used to automate the execution of time-consuming simulations, but its application extends far beyond that. It may be used to process results, or to implement a routine that applies sequential changes to the network and calls PowerFactory’s analysis functions in each step.

1.3 Cases: Smart Grid Application Using PowerFactory

This section is dedicated to show few examples of the use of PowerFactory functionalities to cope with the modelling and simulation requirement of the smart grids.

- Virtual Control Commissioning: Connection to SCADA system via OPC protocol;
- Direct connection to Modbus TCP devices;
- Exporting/importing grids to/from Google Earth using the API.

This section presents only a few successful cases of the use of DIgSILENT PowerFactory for smart grid studies. The reader is invited to visit the DIgSILENT knowledge database in Internet (<http://faq.digsilent.de/powerfactory.html>) where more cases can be found.

1.3.1 *OPC—Virtual Commissioning*

Emulation for Logic Validation is also referred to as *virtual commissioning* (VC). The VC is the process that involves replicating the behaviour of one or more elements of hardware with a software environment (typically for a system under design or further testing).

The objective of the virtual commissioning engineer is to create an environment that mimics the real automation hardware [22].

The final goal of emulation is to provide an environment for the manufacturing automation control engineer to validate their PLC (programmable logic controller) ladder logic and HMI (Human–Machine Interface) files prior to system debug in the plant environment therefore improving quality and enabling a seamless transition from the virtual to physical environment.

Considering the implementation of the commissioning process, there are four different approaches [22]: (i) real commissioning involving a real plant and a real controller; (ii) VC (hardware-in-the-loop commissioning, HIL) involving a virtual

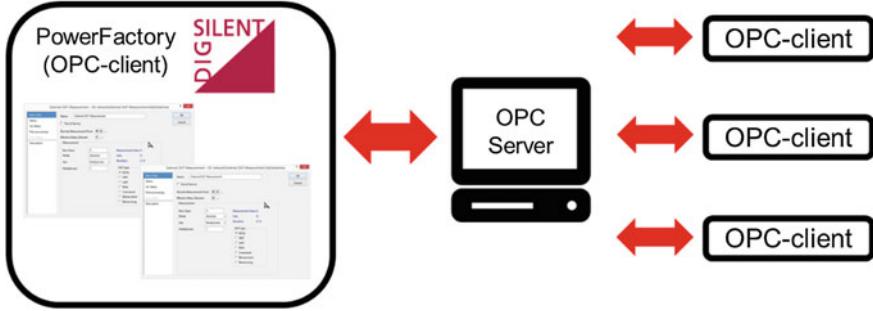


Fig. 1.2 Overview of the OPC interface server/client using OPC

plant and a real controller; (iii) reality-in-the-loop commissioning involving a real plant and a virtual controller; and (iv) constructive commissioning involving a virtual plant a virtual controller. Commissioning engineers often focus on the VC and the constructive commissioning requiring a virtual plant instead of a real plant [22].

This subsection is dedicated to present an illustrative case of the Virtual Control Commissioning using DIgSILENT PowerFactory. The specific example is taken from the presentation of DIgSILENT Ibérica S.L. during the 3rd Southern African DIgSILENT User Conference [23].

PowerFactory uses OPC² (Object Linking and Embedding for Process Control) technology to communicate between the simulation and the power system control on the other side. The Object Linking and Embedding (OLE) for Process Control (OPC) is the name of the standard specification developed by industrial automation task force [23].

The OPC standard specifies the communication of real-time plant data between control devices from different manufacturers. Since it is one of the widely used standards in the control system and SCADA industry, PowerFactory chooses to support OPC standard to communicate with other systems interfacing with simulations running in PowerFactory. This OPC implementation assumes that the PowerFactory software is executed as an OPC client, while the OPC server is controlled via the external source (an illustrative example of the implementation is shown in Fig. 1.2).

The OPC server libraries are obtainable from numerous manufacturers. An example of a freeware OPC server is Matrikon: “MatrikonOPC Simulation Server” [24]. MatrikonOPC provides equipment data connectivity products based on the OPC standard.

²The acronym “OPC” comes from “OLE (Object Linking and Embedding) for Process Control”. Since OLE is based on the Windows COM (Component Object Model) standard, under the hood OPC is essentially COM. Over a network, OPC relies on DCOM (Distributed COM), which was not designed for real-time industrial applications and is often set aside in favour of OPC tunnelling.

OPC interface of PowerFactory provides a very popular interface for industrial networks for SCADA and control system implementation. OPC interfacing requires a dedicated *OPC server* and PowerFactory to run as an *OPC client*. The frequency of data transfer is user defined [25], and the communication between PowerFactory and the OPC server is based on the use of external measurement blocks (*StaExtDatmea*).

The VC can be implemented using real application or final implementation tested in real time and full scale but with simulated signals.

DIgSILENT Ibérica S.L. has successfully used the VC approach in the Spanish system (see details in the document titled “*Technical Requirements for Wind and Photovoltaic Power Plants. DIgSILENT Testing Techniques for Power Plants*” [25]).

A wind farm-centralized voltage controller is tested before commissioning the system as in a real wind farm.

Figure 1.3 shows the schematic diagram showing open connection of PowerFactory with an OPC server to perform real-time simulations. On one side, the control system is programmed in C++ and compiled as a dll; on the other side, the virtual power plant model (wind generators, lines, and transformers) is implemented and simulated in PowerFactory and in between an OPC server.

MATLAB has an OPC toolbox that enables it to connect to an OPC server. The OPC Toolbox™ [26] provides access to live and historical OPC data directly from MATLAB® and Simulink®. Figure 1.4 illustrates a possible PowerFactory/MATLAB coupling scheme [27].

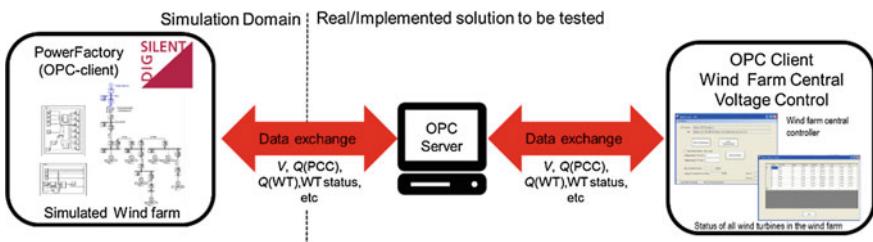


Fig. 1.3 General overview of the OPC—virtual commissioning: example used by wind farm central voltage controller

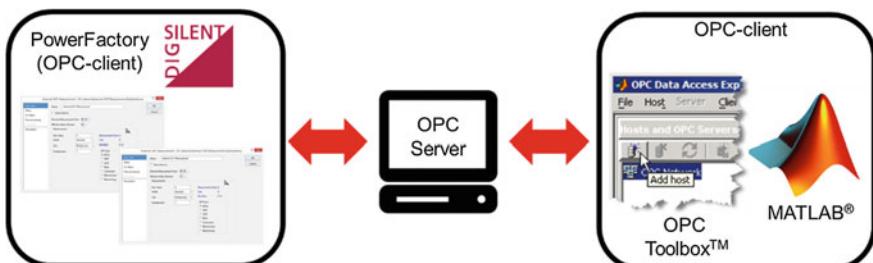


Fig. 1.4 MATLAB and PowerFactory communicate as OPC client

The framework for the co-simulation of power networks and their components including DIgSILENT/PowerFactory and MathWorks MATLAB/Simulink via OPC is well documented in the scientific paper titled “*Framework for Co-Ordinated Simulation of Power Networks and Components in Smart Grids Using Common Communication Protocols*” [19]. Andren et al. use the approach for the co-simulation of a vanadium redox flow battery developed in MATLAB/Simulink which is combined with a grid simulation.

1.3.2 Modbus TCP

Modbus is a serial communication protocol originally published by Modicon (now Schneider Electric [28]) in 1979 for use with its programmable logic controllers (PLCs). Simple and robust, it has since become a de facto standard communication protocol and is now a commonly available means of connecting industrial electronic devices [29].

In a standard Modbus network, there is one master and up to 247 slaves, each with a unique slave address from 1 to 247. The master can also write information to the slaves. The official Modbus specification can be found at <http://www.modbus-ida.org/>.

Modbus is used in multiple master–slave applications to monitor and program devices; to communicate between intelligent devices and sensors and instruments; to monitor field devices using PCs and HMIs.

The Modbus message could be transmitted with a TCP/IP wrapper and sent over a network instead of serial lines, Modbus TCP. Modbus TCP/IP has become ubiquitous because of its openness, simplicity, low-cost development, and minimum hardware required to support it. It is used to exchange information between devices, monitor, and program them. It is also used to manage distributed I/Os, being the preferred protocol by the manufacturers of this type of devices.

The Modbus TCP opens a new dimension for the communication between DIgSILENT and other devices and equipment, allowing so many functionalities as monitor, supervision, control, testing, commissioning, etc.

The DIgSILENT PowerFactory interface with Modbus TCP can be implemented using several approaches: C++ dll—digexdyn, digexfun, digfundpl.

Using C/C++, external DLS functions can be written and then included into DSL blocks to be used in PowerFactory simulation environments [21]. Two main approaches can be used by external *dynamic linking libraries* (.dll).

Figure 1.5 shows a general overview of the implementation to access PowerFactory using an “*event triggered external functions*” in *digexdyn.dll* and the access by *function call* to *digexfun.dll* (more details about this implementation can be found on DIgSILENT Technical Reference “*External DPL Functions*”).

Documentation regarding the external event-driven C/C++ function can be found in the new releases of DIgSILENT PowerFactory. An example for the

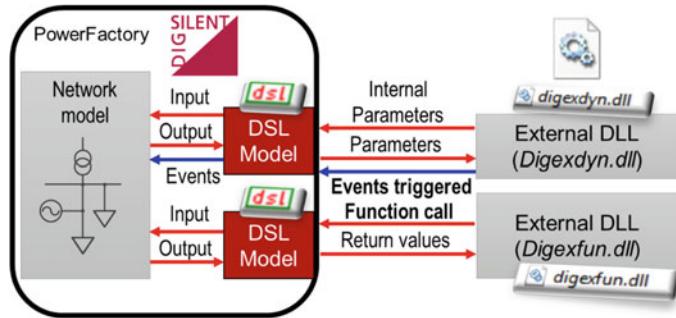


Fig. 1.5 Implementation of external event-driven function and function call in external dynamic linking library (.dll)

user-defined DSL function can be found in the installation directory of PowerFactory under the folder *digexfun*.

The implementation of the libraries is extremely easy. The libraries are placed in the PowerFactory installation folder and automatically loaded (only) at start-up.

Note that multiple libraries can be provided if they use many measurements and inject into simulation (load-flow and RMS/EMT).

Figure 1.6 shows an implementation of Modbus communication protocol based on TCP which is communicated to DIgSILENT PowerFactory using external event-driven function and function call in external dynamic linking library (.dll). DIgSILENT Ibérica S.L. has successfully used this implementation (C++ dll—*digexdyn*, *digexfun*, *digfundpl*) in different contexts: collecting data from SCADA systems and protection devices (read/write registers/coils) [23], etc.

The Modbus communication protocol can be used together with PowerFactory to produce simulation results based on the real status of the power system.

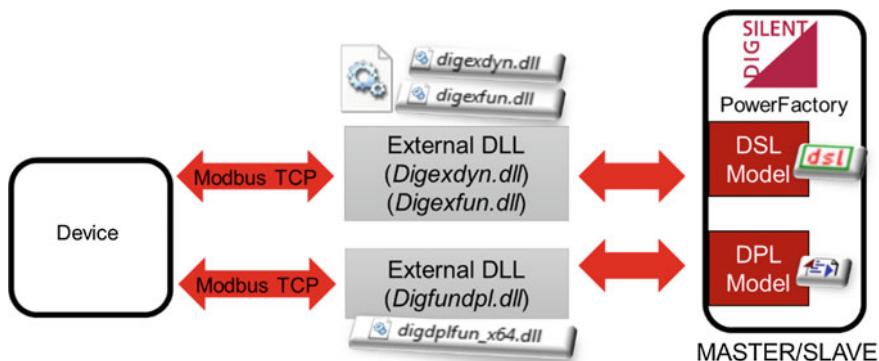


Fig. 1.6 Implementation of external event-driven function and function call in external dynamic linking library (.dll) using Modbus TCP

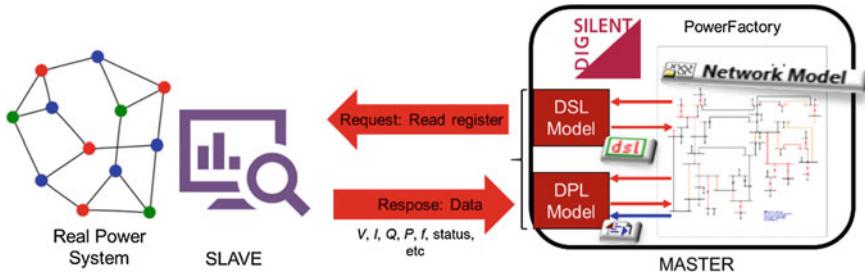


Fig. 1.7 Implementation of Modbus communication protocol for in-loop simulation using DLL C++, direct communication with PowerFactory

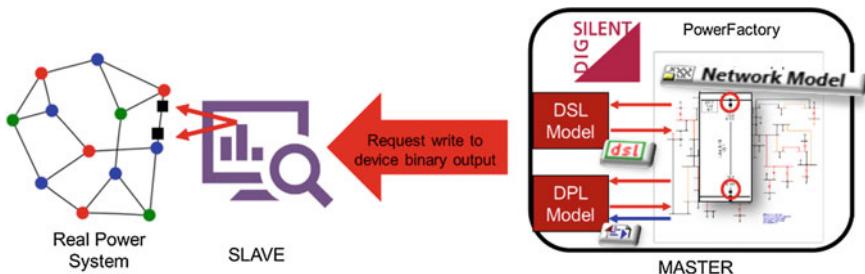


Fig. 1.8 Implementation of Modbus communication protocol for in-loop simulation using DLL C++, direct communication with PowerFactory for operation commands: example showing opening circuit breaker

DIgSILENT Ibérica S.L. has successfully used this approach during one of the stages in the virtual commissioning of a Spanish wind farm [25], combining data collected from SCADA and the voltage controller implemented using C++ dll.

Figure 1.7 shows a general implementation of Modbus communication protocol to collect the data of a real power system and use the collected data to run simulations (load low, RMS, etc.) inside the DIgSILENT PowerFactory. The use of the Modbus communication is not limited to collecting data during simulations; in fact, the communication link could be used to implement operational strategies, e.g. open/closing circuit breakers (see Fig. 1.8).

1.3.3 Geographic Information Systems: Google Earth

Smart grids require not only smart tools to automatize grid operation and planning processes but also require smart approaches to maximize utilization of the capabilities of those tools as well [30].

There are several computer programs that render a simulacrum of the Earth based on satellite imagery. One of them is Google Earth, and it maps the Earth by the superimposition of images obtained from satellite imagery, aerial photography, and *geographic information system* (GIS) onto a 3D globe. The common availability of geospatial data has allowed the power industry to take the advantages of implementing geo-based systems that make data available online, for querying, visualization, for analysis, for updating, and expansion. The applications of the geospatial data in the power systems have evolved from the basic coordinate data of the customers until more sophisticated early instability detection systems.

GISs provide an integrated suite of software for visualizing the system data, tools for network optimization, and a range of automation and information processing systems which assist in the operation, maintenance, and planning of distribution networks. Interfacing of GIS data with sophisticated power system simulation tools facilitates model-updating process for the network planner who needs updated network data for operational and planning analysis.

The GIS together with *Asset Management Systems* (AMSS) is the main sources of network topology and equipment data. DIgSILENT PowerFactory has a powerful name DGS. DGS stands for DIgSILENT Interface for Geographical Information Systems. The DGS is the PowerFactory's standard bidirectional interface specifically designed for bulk data exchange with other applications such as GIS and SCADA.

The DGS can be seen as a file format to exchange data with other data sources; one of them is the GIS, but the potential use can be extended to other simulation applications.

As a format, the DGS can be a pure ASCII-based, but also XML or even in Microsoft Excel or Access format. As a consequence, it is possible to export and import full network models together with element type definitions and graphic representations [21, 31]. Many utilities use GIS exports as a basis for the PowerFactory network model. These exports may comprise detailed substation data including topology, line/cable data, load/generation data and GPS coordinates/schematic diagram information.

Additionally, the DGS allows the results to be exported or load characteristics imported. As a consequence, it is possible to use the DGS to make changes in the network (e.g. breaker and switch position) and perform simulation and analysis automated from external applications (e.g. via RCOM commands).

As Google Earth provides a well-developed and managed geospatial data source, it allows the creation of powerful GIS in combination with DIgSILENT PowerFactory; a general schematic representation of the interaction is shown in Fig. 1.9. PowerFactory is equipped with several high-level functions to allow Compare and Merge Tool, and the versioning mechanism perfectly supports the frequent data exchange with GIS. Also, PowerFactory engines are directly integrated into GIS systems providing calculation functionality such as evaluation of renewable generation connected to the low voltage grid.

Başkent DISCO has successfully developed controllable and integrated enterprise GIS which meets the power utilities requirements. DISCO created an interface

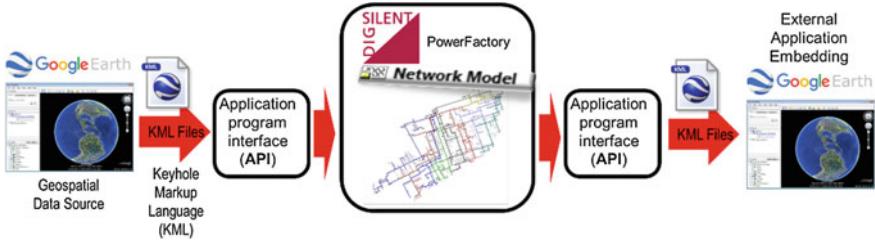


Fig. 1.9 Implementation of import/export Google Earth data using an API and DIgSILENT PowerFactory

to connect ArcGIS database and DIgSILENT PowerFactory software, and it allows using ArcGIS to capture updated grid data along with geographical information for power system analysis with DIgSILENT PowerFactory; more details of this application can be found in [30].

DIgSILENT Ibérica S.L. has developed a user-friendly interface to visualize modal analysis results in Google Earth (more details of this implementation can be found in [32]). The controllability, the observability, and the participation factor of the recorded state variables for each mode and synchronous generator in the project can be displayed on the map. Results are displayed as a vector where its size depends on the magnitude and its orientation depends on the angle.

The core of interfacing DIgSILENT PowerFactory and third party software (i.e. geospatial software or any other) is the development of the appropriate *application program interface* (API). It offers the possibility to embed PowerFactory functionality into their own program, providing direct access to the PowerFactory data model, and gives access to the varied calculations and its results.

The DIgSILENT PowerFactory application programming interface (API) is a functionality available since the version 15.0. The API is designed as a mechanism of automation; as a consequence, the implementation of this interface requires a deep knowledge and understanding of the PowerFactory data model and how to achieve certain tasks manually, including knowledge about the participating objects and commands. The reader must be warned, and the API does not provide a pure calculation engine which can be fed with an abstract calculation topology. PowerFactory is still working as a powerful real power system analysis software with the API.

The PowerFactory API is a logical layer on top of the PowerFactory application that encapsulates the internal data structures and makes them available to external applications (see Figs. 1.10 and 1.11).

An example of the special specific case of integrating smart metres' data in network planning thanks to the API. LINKY project is an example of smart grid project based on PowerFactory, and the project is developed in collaboration with the ERDF (French DNO). ERDF “LINKY” project aims at creating value from the large amount of data from smart metres in terms of distribution network planning and power quality improvement (for more details, see [33]).

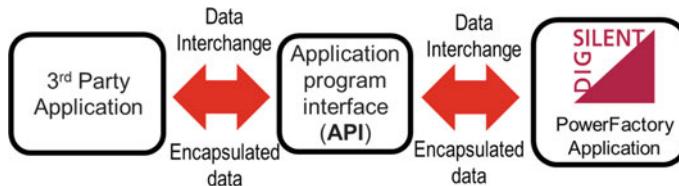


Fig. 1.10 Overview of API interface data flow interaction with DIgSILENT PowerFactory and third party software

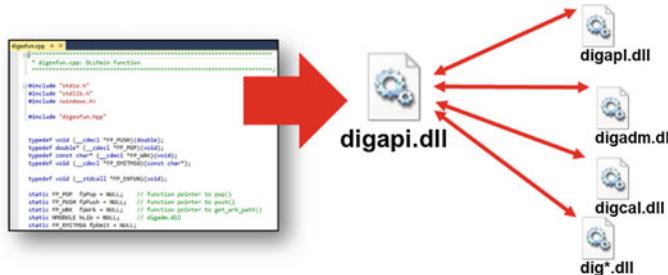


Fig. 1.11 Overview of API interface data flow between .dll

Simulation results of DIgSILENT PowerFactory can be exported into an external application as GIS using KML files. Keyhole Markup Language (KML) is an XML notation for expressing geographic annotation and visualization within Internet-based, two-dimensional maps and three-dimensional Earth browsers. The KML format is designed for GIS application; as a consequence, there is no electrical data information related in this format; however, using an additional property fields allows to export other kind of date in the KML file, e.g. electrical data. DIgSILENT has worked extremely hard to allow the user to define GIS using PowerFactory; as a consequence, several features have been enabled, e.g. define a format/properties to be exported. Since PowerFactory version Version 15.0, the DIgSILENT database has dedicated fields to store the GPS data of bus and lines.

The use of GIS integration to DIgSILENT PowerFactory allows to use new sources of information such as: GIS, SCADA—history, SCADA—real-time operation, weather information (e.g. temperatures, wind speeds), load/generation models, smart metres, market. As a consequence, a new source of large quantity of data is available to add value to the power system planning, operation, and control. The integration of the GIS and PowerFactory can be used in new applications as: early identification of potential network problems (e.g. transmission congestion and bottlenecks, early instability identification), near real-time corrective actions to improve the quality of the supplied service and customer satisfaction, post-event analysis (post-mortem analysis or forensic engineering), improved system in the loop-control design, near real-time power system modelling, and adaptive protection settings.

1.4 Conclusions

The importance of modelling techniques and simulation methods for the development of smart grid systems and applications has already been stated as a very important topic [34].

The existing approaches and tools for power systems and device simulation can be divided into offline steady state, offline transient, and real-time simulation. However, the communication and control models—which are a very important factor for Smart Grid systems, modelling of the ICT systems requires discrete event simulation and therefore need different modeling and simulation approaches. It is evident that the individual domains covered by smart grids—power grid, communication system, controllers—require development of more flexible power system analysis software to cope with the modelling and simulation challenges of the smart grid.

The implementation of highly realistic real-time, massive, online, multi-time frame simulations is required. It must follow a common vision of smart grid functionality, including the agreed vision among politicians, regulators, managers, operators, engineers, and technicians.

The development, testing, and deployment of smart grids will be accelerated by the availability of massive real-time open architecture simulators.

DIgSILENT PowerFactory has a wide spectrum of features that allow the interactions with many systems and add real value to smart grid data, not only in operation but also in network planning.

A summary of the main features oriented to take the maximum values of the smart grids using DIgSILENT PowerFactory has been presented in the previous sections.

The DIgSILENT PowerFactory capabilities can be extended way beyond the other power system analysis software competitors. Three main capabilities deserved special mention:

- (i) OPC allows connecting to the SCADA systems and using PowerFactory. This approach allows the use of the SCADA system as data source providing generate signals for the full-scale system and/or import measurements into power system simulations.
- (ii) DLL (*digexdyn*, *digexfun*, *digdplfun*) allows creating DSL and DPL functions to directly communicate with devices using different standardized protocols (Modbus, DNP3, EtherCat, etc.—this list is increasing).
- (iii) API allows modifying PowerFactory objects from your external applications and sending data to external data repositories.

The aforementioned capabilities allow the use of the full potential of PowerFactory connectivity to fulfil the growing requirements of the smart grids planning and operation.

The smart grid concept is evolving quite fast but DIgSILENT Team is keeping up; as a consequence, more developments and features are expected in short and midterms to enhance much more capability of PowerFactory to cope with the challenges of the future power systems.

References

1. S. Massoud Amin, B.F. Wollenberg, Toward a smart grid: power delivery for the 21st century. *IEEE Power Energy Mag.* **3**(5), 34–41 (2005)
2. English | Oxford Dictionaries. [Online]. Available: <https://en.oxforddictionaries.com/english>. Accessed 29 Jun 2017
3. Smart Grid Definitions of Functions 1 Function Definition. [Online]. Available: https://www.smartgrid.gov/files/definition_of_functions.pdf. Accessed 30 Jun 2017
4. S.A.A. Kazmi, M.K. Shahzad, A.Z. Khan, D.R. Shin, Smart distribution networks: a review of modern distribution concepts from a planning perspective. *Energies* **10**(4) (2017)
5. Power Systems Analysis Software—Open Electrical. [Online]. Available: https://wiki.openelectrical.org/index.php?title=Power_Systems_Analysis_Software
6. MATLAB—MathWorks. [Online]. Available: <https://www.mathworks.com/products/matlab.html>
7. Welcome to Python.org. [Online]. Available: <https://www.python.org/>. Accessed 30 Jun 2017
8. PowerFactory—DIgSILENT Germany. [Online]. Available: <http://www.digsilent.de/index.php/products-powerfactory.html>
9. MatDyn—Electa. [Online]. Available: <http://www.esat.kuleuven.be/electa/teaching/matdyn/>
10. R.D. Zimmerman, C.E. Murillo-Sánchez, R.J. Thomas, MATPOWER: steady-state operations, planning, and analysis tools for power systems research and education. *IEEE Trans. Power Syst.* **26**(1), 12–19 (2011)
11. OpenDSS download | SourceForge.net. [Online]. Available: <https://sourceforge.net/projects/electricdss/>
12. PSAT. [Online]. Available: <http://faraday1.ucd.ie/psat.html>
13. Home—ASPEN, Inc. [Online]. Available: <http://www.aspeninc.com/web/>
14. CYME International Inc.—Home. [Online]. Available: <http://www.cyme.com/>
15. ETAP | Electrical Power System Analysis Software | Power Management System. [Online]. Available: <https://etap.com/>
16. IPSA Power | Software for Power Systems. [Online]. Available: <http://www.ipsa-power.com/>
17. Power Analytics Corp—Power System Design and Optimization. [Online]. Available: <http://www.poweranalytics.com/>
18. Power Technologies International (PTI)—Digital Grid—Siemens
19. F. Andren, M. Stifter, T. Strasser, D. Burnier de Castro, Framework for co-ordinated simulation of power networks and components in smart grids using common communication protocols, in *IECON 2011—37th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2700–2705 (2011)
20. H. Lin, S. Sambamoorthy, S. Shukla, J. Thorp, L. Mili, Power system and communication network co-simulation for smart grid applications, in *ISGT 2011*, pp. 1–6 (2011)
21. M. Stifter, F. Andrén, R. Schwalbe, W. Tremmel, *Interfacing PowerFactory: Co-simulation, Real-Time Simulation and Controller Hardware-in-the-Loop Applications* (Springer International Publishing, Berlin, 2014), pp. 343–366
22. C.G. Lee, S.C. Park, Survey on the virtual commissioning of manufacturing systems. *J. Comput. Des. Eng.* **1**(3), 213–222 (2014)
23. D. Ibérica, S.L.X. Robe, PowerFactory API and smart grid applications interfacing with Google Earth and other systems, in *3rd Southern African DIgSILENT User Conference* (2013)
24. OPC Server from MatrikonOPC—Modbus and 500 OPC Servers and Products. [Online]. Available: <http://www.matrikonopc.com/?gclid=CMQq3ZOG7dQCFRBsGwodANkEYA>. Accessed 03 Jul 2017
25. Technical Requirements for Wind and Photovoltaic Power Plants. Digsilent Testing Techniques for Power Plants
26. OPC Toolbox—MATLAB. [Online]. Available: <https://www.mathworks.com/products/opc.html>. Accessed 03 Jul 2017

27. A. Latif, M. Shahzad, P. Palensky, W. Gawlik, An alternate PowerFactory Matlab coupling approach, in *2015 International Symposium on Smart Electric Distribution Systems and Technologies (EDST)*, pp. 486–491 (2015)
28. Global Specialist in Energy Management and Automation | Schneider Electric. [Online]. Available: <http://www.schneider-electric.co.uk/en/>. Accessed 03 Jul 2017
29. B. Drury, Institution of Electrical Engineers, *The Control Techniques Drives and Controls Handbook* (Institution of Engineering and Technology, 2009)
30. M.E. Cebeci, O.B. Tor, S.C. Yelmaz, O. Gurec, O. Benli, Lessons learnt from interfacing ArcGIS and DIgSILENT powerfactory at Başkent DISCO, in *2016 4th International Istanbul Smart Grid Congress and Fair (ICSG)*, pp. 1–5 (2016)
31. M. Stifter, R. Schwalbe, F. Andren, T. Strasser, Steady-state co-simulation with PowerFactory, in *2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MCPES)*, pp. 1–6 (2013)
32. DIgSILENT Ibérica—Trends in Modal Analysis: Visualization in Google Earth. [Online]. Available: <http://www.digsilentiberica.es/noticia/16/Trends-in-Modal-Analysis-Visualization-in-Google-Earth/>. Accessed 04 Jul 2017
33. G. Roupioz, X. Robe, F. Gorgette, First use of smart grid data in distribution network planning, in *22nd International Conference and Exhibition on Electricity Distribution (CIRED 2013)*, p. 0609 (2013)
34. R. Podmore, M.R. Robinson, The role of simulators for smart grid development. *IEEE Trans. Smart Grid* **1**(2), 205–212 (2010)

Chapter 2

Python Scripting for DIgSILENT PowerFactory: Leveraging the Python API for Scenario Manipulation and Analysis of Large Datasets



Claudio David López and José Luis Rueda Torres

Abstract The need to set up and simulate different scenarios, and later analyse the results, is widespread in the power systems community. However, scenario management and result analysis can quickly increase in complexity as the number of scenarios grows. This complexity is particularly high when dealing with modern smart grids. The Python API provided with DIgSILENT PowerFactory is a great asset when it comes to automating simulation-related tasks. Additionally, in combination with the well-established Python libraries for data analysis, analysis of results can be greatly simplified. This chapter illustrates the synergic relationship that can be established between DIgSILENT PowerFactory and a set of Python libraries for data analysis by means of the Python API, and the simplicity with which this relationship can be established. The examples presented here show that it can be beneficial to exploit the Python API to combine DIgSILENT PowerFactory with other Python libraries and serve as evidence that the possible applications are mainly limited by the creativity of the user.

Keywords Simulation management · Data analysis · Python · Scripting

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_2) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

C. D. López (✉) · J. L. Rueda Torres

Department of Electrical Sustainable Energy, Delft University
of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: C.D.Lopez@tudelft.nl

J. L. Rueda Torres

e-mail: J.L.RuedaTorres@tudelft.nl

2.1 Introduction

Having to simulate a base scenario and many variations of the base scenario is a common occurrence in power system studies. The constantly increasing complexity of smart grids has incremented the number of scenarios that need to be considered, which in turn has caused that larger sets of results need to be analysed. Creating and simulating these scenarios and later comparing and analysing the obtained simulation results can be a repetitive and tedious task that is better carried out with the help of some automation mechanism, such as a scripting language.

Scripting languages are programming languages that rely on a run-time environment for their execution. The run-time environment translates—at run-time—the tasks stated in a script into instructions that a computer can execute. Since scripts are not compiled before execution, in some cases there may be performance drawbacks. However, scripting languages are well suited for automating sequences of high-level tasks that would otherwise be executed manually by the user, making them a natural choice for automating simulation-related tasks.

One of the two scripting languages supported by PowerFactory is the DIgSILENT Programming Language (DPL). DPL is a scripting language developed by DIgSILENT exclusively for use with PowerFactory, intended for automating repetitive tasks and processing results. It has a syntax similar to C, and aside from supporting variable definitions, assignments, program flow control and method calls, it also implements a set of mathematical expressions and functions to simplify calculations.

Since version 15.1, PowerFactory also includes a Python Application Programming Interface (API) that defines a set of functions and objects that make almost the entire functionality that PowerFactory implements accessible from a Python script, allowing the integration of PowerFactory into other Python applications as well. Unlike DPL, the Python programming language is an open source, general-purpose scripting language and one of the most popular programming languages of current times. Known for its versatility and for facilitating the production of readable code, Python is a common pick for applications where functionality is to be achieved with minimum programming effort. It has a much clearer syntax than DPL, and fewer lines of code are typically required in a Python script to replicate the functionality of a DPL script. In comparison, Python scripts are easier to write, read, debug and maintain than DPL scripts.

Some additional Python features that are worthy of note are its multi-paradigm nature, as it fully supports the object-oriented and structured paradigms, while partially implementing others such as functional programming, and its extensive standard library that enables ‘out-of-the-box’ use in many programming projects. In addition, one of the most salient traits that make Python appealing in the context of power system analysis is its popularity within the scientific and engineering communities [1]. This popularity has materialized in a vast collection of open source libraries for scientific computing, which in combination with PowerFactory can be a great asset. Some of the most mature and extensively used libraries for scientific and engineering applications are NumPy [2] and SciPy [3] (extended mathematical functionality), Pandas [4] (manipulation and analysis of large datasets), scikit-learn [5] (machine learning) and Matplotlib [6] (data visualization), among many others.

This chapter illustrates the synergy that can be established between PowerFactory and a set of Python libraries for data analysis, and the simplicity with which this synergy can be established. A series of example Python scripts that build upon each other are used to illustrate this synergy. An archive that includes all of these scripts and a PowerFactory project to test them accompanies this chapter as supporting material. Each Python script discussed in this chapter indicates the name of the file in the archive where the code can be found.

This chapter begins by introducing the Python API through an example Python class called *PowerFactorySim*, which implements methods for starting PowerFactory, activating a project, setting up simulations and running them. It then exemplifies how simulation scenarios can be manipulated and controlled by extending the functionality of the *PowerFactorySim*. Finally, the data analysis capabilities of Python are displayed by analysing the results of the examples introduced in previous sections.

2.2 The Python API

The Python API provided with PowerFactory makes the interaction between any Python script and PowerFactory possible, as illustrated in Fig. 2.1. The only pre-requisite to using the API is that a compatible Python interpreter must be available. In the case of PowerFactory 15.2, an interpreter for Python 3.4 may be used. The examples presented in this chapter are developed for these versions.¹

The Python API is implemented in the *powerfactory.pyd* Python dynamic module located in PowerFactory's installation folder. This section shows how to import this module into a Python script and how to make use of the Python API implemented in it. In order to do so, the basic methods of the *PowerFactorySim* example class are introduced.

2.2.1 *PowerFactory in Engine Mode*

In engine mode, PowerFactory can be imported to and run from an external Python script, without the need for a graphical user interface. This mode makes it possible to seamlessly integrate PowerFactory functionality into any Python program and vice versa. To run PowerFactory in engine mode, neither the graphical user interface nor any other instance of PowerFactory may be running.

To import the *powerfactory.pyd* dynamic module into a Python script, the location of the module must be added to the Python path, so the Python interpreter becomes aware of its existence. The location of this module can be added with the

¹Detailed installation instructions for the Python interpreter are provided in Chap. 19 of the PowerFactory 15 user manual (DIgSILENT PowerFactory Version 15 User Manual, DIgSILENT GmbH, Gomaringen, Germany, 2015). See Sect. 4.1 of this chapter for a discussion on how to manage multiple versions of the Python interpreter simultaneously.

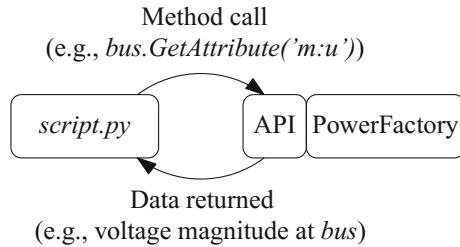


Fig. 2.1 Interaction between a Python script and PowerFactory through the Python API

```

1 import sys
2 sys.path.append(
3     r"C:\Program Files\DIgSILENT\PowerFactory 15.2\Python\3.4")
4 import powerfactory as pf

```

Fig. 2.2 Importing PowerFactory in engine mode (*pfsim.py* file)

code shown in Fig. 2.2, which must be at the header of the script. Line 3 in Fig. 2.2 is the path to the *powerfactory.pyd* module, which must be adjusted according to each individual PowerFactory installation.

Once the dynamic module has been imported, the *pf.GetApplication* method starts PowerFactory in engine mode.

2.2.2 Activating a Project and a Study Case

In the *PowerFactorySim* example class, projects and study cases are activated in the *__init__* method, which is called automatically when an object of this class is created.² The implementation of the *__init__* method is shown in Fig. 2.3. In this method, line 5 starts PowerFactory in engine mode, lines 7 and 8 activate the project specified in the *project_name* argument and lines 10–14 activate the study case specified in the *study_case* argument. The *folder_name* argument can be used if the project is located inside a folder in the PowerFactory database, and otherwise, it can be ignored.

Note that the *os* module from the Python standard library is used in line 8 to safely create the path to the project location within the PowerFactory database. This path is created by concatenating the *folder_name* and *project_name* arguments. To make the *os* module accessible, the line *import os* must be added to the header of the *pfsim.py* file.

²In Python, the *__init__* method of a class is automatically called when the class is instantiated. The line *pfsim = PowerFactorySim()* causes the *__init__* method from Fig. 3 to be called.

```

1  class PowerFactorySim(object):
2      def __init__(self, folder_name='', project_name='Project',
3                   study_case_name='Study Case'):
4          # start PowerFactory
5          self.app = pf.GetApplication()
6          # activate project
7          self.project = self.appActivateProject(
8              os.path.join(folder_name, project_name))
9          # activate study case
10         study_case_folder = self.app.GetProjectFolder('study')
11         study_case = study_case_folder.GetContents(
12             study_case_name+'.IntCase')[0]
13         self.study_case = study_case[0]
14         self.study_case.Activate

```

Fig. 2.3 Starting PowerFactory in engine mode and activating a project and a study case (*PowerFactorySim* class, *pfsim.py* file)

2.2.3 *Running Simulations and Accessing Results*

The workflow of a typical simulation can be divided into three stages: preparation, execution and result retrieval. These three stages can be implemented as three methods in the *PowerFactorySim* example class. The implementation of these three methods varies between static and dynamic simulations, since the parameters that need to be specified while preparing a simulation are different between static and dynamic cases, and both the nature of the results and the way they are retrieved differ.

2.2.3.1 *Static Simulations*

PowerFactory is capable of running many different static simulations, such as load flow and short circuit analyses. Figure 2.4 shows methods to prepare and run a load flow simulation and then get the results.

In the *prepare_loadflow* method, line 5 retrieves the *ComLdf* load flow object and line 7 sets the load flow mode. The mode can be selected through the *ldf_mode* argument, and it can either be ‘*balanced*’, ‘*unbalanced*’ or ‘*dc*’, with the default mode being ‘*balanced*’.

The *run_loadflow* method only calls the *Execute* method in the load flow object. Its return values are *False* if the load flow calculation is successful and *True* if it is not (e.g. non-convergence).

The *get_bus_voltages* method exemplifies how simulation results can be accessed. In this method, line 15 retrieves all *ElmTerm* terminal objects and stores them in the *buses* list. Then, line 18 retrieves the voltage magnitude of each bus and stores it in a dictionary as the one shown in Fig. 2.5. The *loc_name* attribute used in line 18, which is a parameter present in all grid components, holds the name of the bus.

```

1 def prepare_loadflow(self, ldf_mode='balanced'):
2     # translate load flow mode keyword to its int equivalent
3     modes = {'balanced': 0, 'unbalanced': 1, 'dc': 2}
4     # retrieve load-flow object
5     self.ldf = self.app.GetFromStudyCase('ComLdf')
6     # set load flow mode
7     self.ldf.iopt_net = modes[ldf_mode]
8
9 def run_loadflow(self):
10    return bool(self.ldf.Execute())
11
12 def get_bus_voltages(self):
13    voltages = {}
14    # collect all bus elements
15    buses = self.app.GetCalcRelevantObjects('* ElmTerm')
16    # store voltage of each bus in a dictionary
17    for bus in buses:
18        voltages[bus.loc_name] = bus.GetAttribute('m:u')
19
20    return voltages

```

Fig. 2.4 Running a power flow and accessing the results (*PowerFactorySim* class, *pfsim.py* file)

```

voltages =
    'Bus1': 1.0,
    'Bus2': 0.99,
    'Bus3': 0.97,
    :
}

```

Fig. 2.5 Structure of the voltages dictionary returned by the *get_bus_voltages* method: keys are bus names, and values are voltage magnitudes

The *get_bus_voltages* method can be easily modified to retrieve other results by replacing the ‘*.ElmTerm’ string with the object of interest (e.g. ‘SM1.ElmSym’ to get the synchronous machine SM1 or ‘*.ElmSym’ to get all synchronous machines) and the ‘*m:u*’ string with the result of interest.

The *GetCalcRelevantObjects* method and a good naming convention for grid elements can be a powerful combination. By using the * placeholder it is possible to retrieve a list of elements whose names match a certain pattern. For example, if all buses include voltage information in their names, it becomes much simpler to retrieve buses by voltage level. If buses are named, for instance, *Bus_20kV_1*, *Bus_20kV_2*, *Bus_20kV_3*, ..., *Bus_230kV_1*, *Bus_230kV_2*, then *GetCalcRelevantObjects* (*‘Bus_20kV* ElmTerm’*) would retrieve all 20 kV buses. This filtering capability can become quite convenient as grids increase in size.

2.2.3.2 Dynamic Simulations

The methods introduced in Fig. 2.6 make it possible to run a dynamic simulation. Preparing a dynamic simulation involves selecting the results that must be monitored, setting the simulation type (RMS or EMT), setting the start and end times as well as the step size and calculating initial conditions. This procedure is implemented in the *prepare_dynamic_sim* method. The variables to monitor are selected with a dictionary in the format of the *MONITORED_VARIABLES* dictionary shown in Fig. 2.7 that must be passed to the method in the *monitored_variables* argument.

This dictionary allows to conveniently define a large number of variables to be monitored by relying on the same variable naming convention used by the *GetCalcRelevantObjects* method. Dynamic results are not stored directly in each grid element, but are instead stored in an *ElmRes* object. This object is retrieved from line 5. Later, lines 7–12 take the variables specified in the *monitored_variables* dictionary and set them for monitoring in the *ElmRes* object. Lines 14 and 15 retrieve the *ComInc* (initial conditions) and *ComSim* (time domain simulation) objects. Additionally, the simulation type is set using the *sim_type* argument. The possible options are ‘rms’ (RMS simulation) and ‘ins’ (EMT simulation). Simulation time constraints are set between lines 16 and 21. Finally, the initial conditions are calculated in line 23.

Similarly to the static case, the *run_dynamic_sim* method executes the dynamic simulation. If the simulation is successful, the method returns *False*; otherwise *True* is returned.

To retrieve the results of a dynamic simulation, it is necessary to understand the structure of the *ElmRes* object. This object organizes results in tabular form, where each column is a time series of a different result variable. Result variables that belong to the same grid element are placed next to each other, as it is shown in Table 2.1. The method *get_dynamic_results* allows retrieving the results stored in the *ElmRes* object. Once the results are loaded as in line 32, the column in the *ElmRes* object that holds the results of interest must be identified. This is done with the *ResGetIndex* method as in line 34. Since columns need to be read element by element in a loop, it is necessary to determine the number of elements in the column, which is done with the *ResGetValueCount* method, as in line 37. Finally, the results can be copied to a list, as shown in lines 39–44. Note that in order to access the time column in line 42, the column index -1 is used in the *ResGetData* method (see Table 2.1 as well).

2.2.4 Example: Running an EMT Simulation

Using the functionality provided by the *PowerFactorySim* example class, the *emt.py* script presented in Fig. 2.7 exemplifies how an EMT simulation can be run. The simulation is executed on the two-area system from [7] (see Fig. 2.25).

```

1 def prepare_dynamic_sim(self, monitored_variables,
2                         sim_type='rms', start_time=0.0,
3                         step_size=0.01, end_time=10.0):
4     # get result file
5     self.res = self.app.GetFromStudyCase('*ElmRes')
6     # select results variables to monitor
7     for elm_name, var_names in monitored_variables.items():
8         # get all network elements that match 'elm_name'
9         elements = self.app.GetCalcRelevantObjects(elm_name)
10        # select variables to monitor for each element
11        for element in elements:
12            self.res.AddVars(element, *var_names)
13    # retrieve initial conditions and time domain sim. objects
14    self.inc = self.app.GetFromStudyCase('ComInc')
15    self.sim = self.app.GetFromStudyCase('ComSim')
16    # set simulation type: 'rms' or 'ins' (for EMT)
17    self.inc.iopt_sim = sim_type
18    # set start time, step size and end time
19    self.inc.tstart = start_time
20    self.inc.dtgrd = step_size
21    self.sim.tstop = end_time
22    # set initial conditions
23    self.inc.Execute()
24
25 def run_dynamic_sim(self):
26     return bool(self.sim.Execute())
27
28 def get_dynamic_results(self, elm_name, var_name):
29     # get network element of interest
30     element = self.app.GetCalcRelevantObjects(elm_name)[0]
31     # load results from file
32     self.app.ResLoadData(self.res)
33     # find colum in results file that holds result of interest
34     col_index = self.app.ResGetIndex(
35         self.res, element, var_name)
36     # get number of rows (points in time) in the result file
37     n_rows = self.app.ResGetValueCount(self.res, 0)
38     # read results and time and store them in lists
39     time = []
40     var_values = []
41     for i in range(n_rows):
42         time.append(self.app.ResGetData(self.res, i, -1)[1])
43         var_values.append(
44             self.app.ResGetData(self.res, i, col_index)[1])
45
46     return time, var_values

```

Fig. 2.6 Running a dynamic simulation and accessing the results (*PowerFactorySim* class, *pfsim.py* file)

```

1 import csv
2 from pfsim import PowerFactorySim
3
4
5 FOLDER_NAME = ''
6 PROJECT_NAME = '2A4G'
7 STUDY_CASE_NAME = 'Study Case'
8 MONITORED_VARIABLES = {
9     'G1.ElmSym': ['s:phi', 's:speed', 's:fe'],
10    '*.ElmTerm': ['m:ul:A', 'm:ul:B', 'm:ul:C']
11 }
12
13 # activate project and study case
14 sim = PowerFactorySim(
15     folder_name=FOLDER_NAME,
16     project_name=PROJECT_NAME,
17     study_case_name=STUDY_CASE_NAME)
18 # prepare EMT simulation
19 sim.prepare_dynamic_sim(
20     monitored_variables=MONITORED_VARIABLES,
21     sim_type='ins',
22     start_time=0.0,
23     step_size=0.0001,
24     end_time=0.02)
25 # run EMT simulation
26 sim.run_dynamic_sim()
27 # retrieve line-to-line volages from one bus
28 t, ula = sim.get_dynamic_results(
29     'Bus_20kV_1.ElmTerm', 'm:ul:A')
30 _, ulb = sim.get_dynamic_results(
31     'Bus_20kV_1.ElmTerm', 'm:ul:B')
32 _, ulc = sim.get_dynamic_results(
33     'Bus_20kV_1.ElmTerm', 'm:ul:C')
34 # store line-to-line volages in csv file
35 with open('res_emt.csv', 'w', newline='') as csvfile:
36     csvwriter = csv.writer(csvfile)
37     csvwriter.writerow(['t', 'ula', 'ulb', 'ulc'])
38     for row in zip(t, ula, ulb, ulc):
39         csvwriter.writerow(row)

```

Fig. 2.7 Running an EMT simulation (*emt.py* file)**Table 2.1** Example of the structure of an *ElmRes* results object

Column number								
Row number	Time	Gen. <i>ElmSym</i>			Line. <i>ElmLne</i>			
		<i>s:phi</i>	<i>s:speed</i>	<i>s:fe</i>	...	<i>u:bus1</i>	<i>i:bus1</i>	...
0	0.00	-2.454891	0.996626	0.996626		0.914305	1.995131	
1	0.01	-2.466818	0.997046	0.997046		0.904337	1.991621	
2	0.02	-2.477128	0.997484	0.997484		0.892943	1.984058	
:				:				.

2.2.4.1 The Script

The *MONITORED_VARIABLES* dictionary in Fig. 2.7 indicates to the *prepare_dynamic_sim* method that rotor angle, speed and electrical frequency are to be monitored for one of the generators, and all three line-to-line voltages must be monitored for every bus in the system. For the sake of simplicity, only the line-to-line voltages of a single bus are retrieved in this example. The results are temporarily stored in the lists *t* (time), *ula*, *ulb* and *ulc* (voltages), and permanently in a CSV³ file.

2.2.4.2 Set-up and Execution

The first step to setting up this example is to import the provided *2A4G.pfd* project into PowerFactory. With the project correctly imported, the user interface must be closed before PowerFactory can be run in engine mode. If the right Python interpreter is already installed, the next step is to open the Windows command line (CMD).

To execute *emt.py*, the current directory must be changed to the directory where the script is located. If, for example, the location is *C:\scripts*, the current directory can be changed by entering the command *cd C:\scripts* in the CMD. The execution is started by entering *python emt.py* in the CMD. This is illustrated in Fig. 2.8. Once the script is executed, the results are stored in the *res_emt.csv* file.

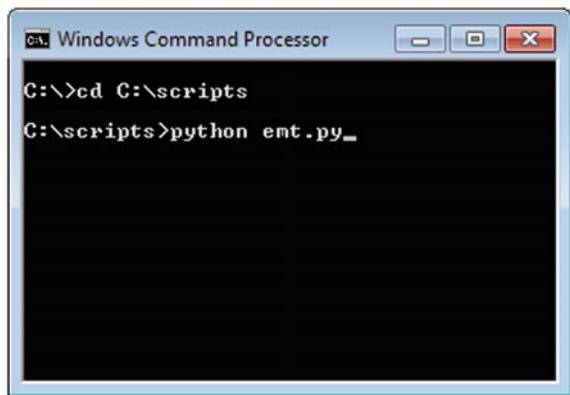
The *emt.py* script can be applied to any other PowerFactory project by simply modifying the constants *FOLDER_NAME*, *PROJECT_NAME* and *STUDY_CASE_NAME* as required.

2.3 Scenario Manipulation with the Python API

One of the main strengths of the Python API is that it provides the user with the ability to automatically generate and simulate a large number of scenarios, without having to manually set them up one by one using the graphical user interface. This section extends the basic functionality of the *PowerFactorySim* example class introduced in Sect. 2.2 to include methods for scenario manipulation.

³CSV (Comma-Separated Values, csv) are plain text that files store data in tabular form by separating values with commas.

Fig. 2.8 Running the *emt.py* script (see Fig. 2.7) located in *C:\scripts* from the Windows Command Prompt (CMD)



2.3.1 Static Scenarios

Variations of a base static scenario can be of many different types; typical are those variations where components are modified (e.g. different line types need to be tested), their power generation or consumption is varied (e.g. probabilistic load flow, quasi-static simulations) and topologies are modified (e.g. grid expansion studies). The first two can be achieved by modifying grid element parameters, while the simplest way to achieve the former is by connecting and disconnecting grid elements.

2.3.1.1 Modifying Parameters Values

The first step to modifying system parameters is identifying their names. With the help of the graphical user interface, names are easily identified by opening the dialogue that belongs to the object of interest. In Fig. 2.9, the dialogue corresponding to an *ElmLne* line object is shown. By hovering the mouse over the parameter of interest, the parameter name is revealed. In the case of the line object, Fig. 2.9 indicates that the parameter name for ‘number of parallel lines’ is *nlnum*.

Once the parameter name has been identified, modifying it is straightforward. Following the example of the line, to change the number of parallel lines from one to two it is sufficient to write *line.nlnum = 2*, where *line* is an object of type *ElmLne*.

The same method can be employed to set power and reactive power consumption for system loads. Figure 2.10 introduces the *set_all_loads_pq* method, which sets active and reactive power values for each load in the system. The arguments of this method are two dictionaries, one for active and one for reactive power. The keys of these dictionaries are load names.

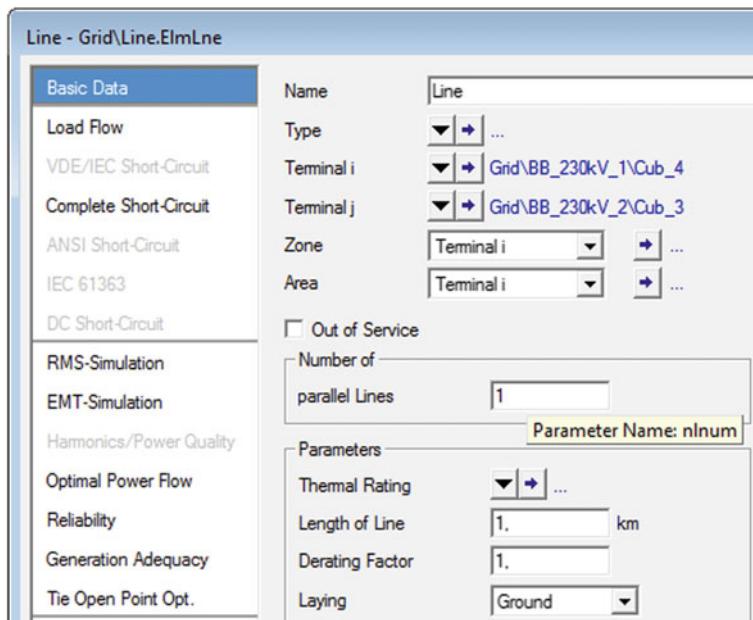


Fig. 2.9 Identifying parameter names in an *ElmLne* line object: hovering the mouse pointer over the ‘number of parallel lines’ field reveals the parameter name *nlnum*

```

1 def set_all_loads_pq(self, p_load, q_load):
2     # collect all load elements
3     loads = self.app.GetCalcRelevantObjects('*.ElmLod')
4     # set active and reactive load values
5     for load in loads:
6         load.plini = p_load[load.loc_name]
7         load.qlini = q_load[load.loc_name]

```

Fig. 2.10 Setting active and reactive power of all loads (*PowerFactorySim* class, *pfsim.py* file)

2.3.1.2 Connecting and Disconnecting Components

When it comes to topological variations, the simplest way is to connect and disconnect components from the grid. This can be achieved either by setting components as out of service or by opening switches. The main difference between both is that switches can also be opened and closed during a dynamic simulation and have additional properties that are useful for transient simulation. Furthermore, since switches are physical elements, they are only present in grid components, whereas other—non-physical—types of objects, such as events, can also be set out of service.

```

1 def toggle_out_of_service(self, elm_name):
2     # collect all elements that match elm_name
3     elms = self.app.GetCalcRelevantObjects(elm_name)
4     # if elm is out of service, switch to in service, else,
5     # switch to out of service
6     for elm in elms:
7         elm.outserv = 1 - elm.outserv

```

Fig. 2.11 Toggling the out-of-service element parameter (*PowerFactorySim* class, *pfsim.py* file)

```

1 def toggle_switches(self, elm_name):
2     # collect all elements that match elm_name
3     elms = self.app.GetCalcRelevantObjects(elm_name)
4     # collect all switches
5     sws = self.app.GetCalcRelevantObjects('*_staSwitch')
6     # find switches corresponding to each elm and toggle them
7     for elm in elms:
8         cubs = elm.GetCubicle(0) + elm.GetCubicle(1)
9         for sw in sws:
10             if sw.fold_id in cubs:
11                 sw.on_off = 1 - sw.on_off

```

Fig. 2.12 Toggling switches (*PowerFactorySim* class, *pfsim.py* file)

A method for toggling the out-of-service state of system elements is shown in Fig. 2.11. This method toggles the *outserv* parameter of all system elements that match *elm_name*.

Similarly to the *toggle_out_of_service* method from Fig. 2.11, the *toggle_switches* method from Fig. 2.12 toggles the *on_off* property of all the switches that reside in the cabinets of the elements selected through the *elm_name* argument. What makes this method more cumbersome than the *toggle_out_of_service* method is that some elements have one switch, while others have two. For example, elements such as loads and generators have only one switch, whereas elements like lines and transformers have two (one on each terminal).

2.3.2 Dynamic Scenarios

Dynamic scenarios can be created the same way static scenarios are, as long as each variation is performed before the simulation is executed. Changes that need to occur while the simulation is running can be implemented through events. PowerFactory allows the creation of events such as dispatch events (*EvtGen*), load events (*EvtLod*), parameter events (*EvtParam*) and short circuit events (*EvtShc*), among many others.

2.3.2.1 Creating Events

Creating an event involves creating an object inside the events folder of the active study case. This can be carried out in three main steps: first, the event folder needs to be retrieved using the *GetFromStudyCase* method. Then, the event object must be created in this folder using the *CreateObject* method. Finally, all relevant object parameters must be set, one by one. This process is exemplified in the *create_short_circuit* method from Fig. 2.13. This method creates a three-phase short circuit that is applied to an element specified in *target_name*, at a time specified in *time* and of an optional duration. If a short circuit duration is specified in the *duration* argument, then two events are created: one short circuit event and one short circuit clearing event. If no duration is specified, the short circuit is never cleared.

2.3.2.2 Deleting Events

All created events are stored in the events folder of the active study case. If a new event is to be tested but older events are no longer required, it is advisable to delete them. Creating an event with the same name as an already existing event does not

```

1  def create_short_circuit(self, target_name, time,
2                           duration=None, name='sc'):
3      # get element where the short circuit will be applied
4      target = self.app.GetCalcRelevantObjects(target_name)[0]
5      # get the events folder from active study case
6      evt_folder = self.app.GetFromStudyCase('IntEvt')
7      # create an empty event of type EvtShc (short circuit)
8      evt_folder.CreateObject('EvtShc', name)
9      # get the newly created event
10     sc = evt_folder.GetContents(name+'.EvtShc')[0][0]
11     # set time, target and type of short circuit (3-phase)
12     sc.time = time
13     sc.p_target = target
14     sc.i_shc = 0
15     # set clearing event if required
16     if duration is not None:
17         # create an empty event of type EvtShc (short circuit)
18         evt_folder.CreateObject('EvtShc', name+'_clear')
19         # get the newly created event
20         scc = evt_folder.GetContents(
21             name+'_clear'+'.EvtShc')[0][0]
22         # set time, target and type of event (clearing)
23         scc.time = time + duration
24         scc.p_target = target
25         scc.i_shc = 4

```

Fig. 2.13 Creating a short circuit event (*PowerFactorySim* class, *pfsim.py* file)

```

1 def delete_short_circuit(self, name='sc'):
2     # get the events folder from active study case
3     evt_folder = self.app.GetFromStudyCase('IntEvt')
4     # find the short circuit and clear event to delete
5     sc = evt_folder.GetContents(name+'.EvtShc')[0]
6     scc = evt_folder.GetContents(name+'_clear'+'.EvtShc')[0]
7     # delete short circuit and clear events if they exist
8     if sc:
9         sc[0].Delete()
10    if scc:
11        scc[0].Delete()

```

Fig. 2.14 Deleting a short circuit event (*PowerFactorySim* class, *pfsim.py* file)

override the older event; the result is that now both events reside in the events folder. This could lead to events being triggered inadvertently. To delete events it is possible to take advantage of the *Delete* method present in all PowerFactory data objects. Figure 2.14 introduces the *delete_short_circuit* method, a method that deletes short circuit events created with the *create_short_circuit* method from Fig. 2.13.

2.3.3 Example with Static Scenarios: Probabilistic Load Flow

This example introduces the *MontecarloLoadFlow* class that provides functionality for running Monte Carlo probabilistic load flows and shows its use in the *prob_lf.py* script. The simulation is executed on the two-area system from [7] (see Fig. 2.25).

2.3.3.1 The Probabilistic Load Flow Class

The *MontecarloLoadFlow* example class introduced in Fig. 2.15 provides methods for executing a Monte Carlo probabilistic load flow with normally distributed loads. This implementation is based on the DPL script from [8]. Two methods are defined in this class: the *gen_normal_loads_pq* method generates normally distributed active and reactive load values, and the *monte_carlo_loadflow* method executes load flows using normally distributed loads. The remaining methods required for the execution of the probabilistic load flow are inherited from the *PowerFactorySim* class (see line 5).

To generate normally distributed loads, the *gen_normal_loads_pq* requires the total active and reactive power consumption in the *p_total* and *q_total* arguments, as well as the active and reactive power consumption of each individual load in the *p_base* and *q_base* arguments, all of them in the base load scenario. The *p_base* and *q_base* arguments are dictionaries, where the keys are load names and the

```

1 import random
2 import warnings
3 from math import sqrt, cos, log, pi
4
5 class MontecarloLoadFlow(PowerFactorySim):
6     def gen_normal_loads_pq(self, p_total, q_total,
7                             p_base, q_base, std_dev=0.1):
8         # generate 2 random numbers from uniform distribution
9         rand1 = random.uniform(0, 1)
10        rand2 = random.uniform(0, 1)
11        # sample normally distributed load
12        p_total_rand = p_total*(
13            1 + std_dev*sqrt(-2*log(rand1))*cos(2*pi*rand2))
14        q_total_rand = q_total*((
15            1 + std_dev*sqrt(-2*log(rand1))*cos(2*pi*rand2)))
16        # collect all load elements
17        loads = self.app.GetCalcRelevantObjects('* ElmLod')
18        # store normally distributed load values in dictionary
19        p_normal = {}
20        q_normal = {}
21        for load in loads:
22            p_normal[load.loc_name] = (p_base[load.loc_name]
23                                       /p_total*p_total_rand)
24            q_normal[load.loc_name] = (q_base[load.loc_name]
25                                       /q_total*q_total_rand)
26
27        return p_normal, q_normal
28
29    def monte_carlo_loadflow(self, n_samples, std_dev,
30                            max_attempts=10):
31        # set up the load flow object
32        self.prepare_loadflow()
33        # get base (initial) load values
34        p_base, q_base = self.get_all_loads_pq()
35        # calculate total base system load
36        p_total = sum(p_base.values())
37        q_total = sum(q_base.values())
38        # sample load flow 'n_samples' times
39        for sample in range(n_samples):
40            # re-attempt load flow in case of non-convergence
41            for attempt in range(max_attempts):
42                # generate random normally distributed loads
43                p_normal, q_normal = self.gen_normal_loads_pq(
44                    p_total, q_total, p_base, q_base,
45                    std_dev=std_dev)
46                # set network loads to random load values
47                self.set_all_loads_pq(p_normal, q_normal)
48                # run load flow with normally dist. loads
49                failed = self.run_loadflow()
50                if failed:

```

Fig. 2.15 Monte Carlo probabilistic load flow class (*MontecarloLoadFlow* class, *pfsim.py* file)

```

51             warnings.warn(
52                 "Sample " + str(sample)
53                 + " didn't converge, re-attempt "
54                 + str(attempt+1) + " out of "
55                 + str(max_attempts))
56         else:
57             break
58
59         # yield load flow results
60         yield self.get_bus_voltages()
61
62     # restore network to base load (initial) values
63     self.set_all_loads_pq(p_base, q_base)

```

Fig. 2.15 (continued)

values are either active or reactive power. The standard deviation of the distribution can also be set, and it is 10% by default. Although it is possible to calculate p_{total} and q_{total} from p_{base} and q_{base} , and to therefore reduce the number of arguments the *gen_normal_loads_pq* method takes, it is not convenient to make this calculation inside the *gen_normal_loads_pq* method, as this would reduce the efficiency of the code.

The *monte_carlo_loadflow* method assumes that the initial state of the grid is the base load scenario. The method starts by preparing the *ComLdf* load flow object and storing the initial load values. Once the total base load has been calculated, the method enters the sampling loop (line 39), where samples from a normal distribution are taken through the *gen_normal_loads_pq*, the normally distributed loads are set and the load flow is calculated. The second loop (line 41) re-attempts a load flow calculation with a different load sample if the previous one does not converge. In case of non-convergence, a warning is issued. The load flow results are yielded in line 59. For simplicity, only voltage magnitudes are retrieved in this case. Finally, the method restores the grid to its initial base load condition.

The *yield* keyword used in line 59 of the *monte_carlo_loadflow* method has a similar purpose to that of the *return* keyword, but it transforms the method into a generator. Generators are iterables,⁴ but they can be iterated over only once because they do not store values in memory; they generate the values at run-time. Methods that return a data structure of potentially large size can be problematic if not enough memory is available. One such method is *monte_carlo_loadflow*, which could produce a large set of results depending on the size of the grid and the number of samples taken from the normal distribution. It is recommended to approach these cases through Python generators instead of traditional functions that return all results at once, and to process them as they are generated [9].

⁴Python iterables are objects capable of returning their members one at a time. For example, the line *for member_object in iterable_object:* makes the iterable *iterable_object* return its members one by one, where *iterable_object* can be a list, a dictionary, etc.

```

1 import csv
2 from pfsim import MontecarloLoadFlow
3
4
5 FOLDER_NAME = ''
6 PROJECT_NAME = '2A4G'
7 STUDY_CASE_NAME = 'Study Case'
8
9 N_SAMPLES = 2500
10 STD_DEV = 0.1 # 0.1 = 10%
11
12 # activate project and study case
13 sim = MontecarloLoadFlow(
14     folder_name=FOLDER_NAME,
15     project_name=PROJECT_NAME,
16     study_case_name=STUDY_CASE_NAME)
17 # create montecarlo load flow iterable object
18 mcldf = sim.monte_carlo_loadflow(N_SAMPLES, STD_DEV)
19 # create a csv file for storing results
20 with open('res_prob_lf.csv', 'w', newline='') as csvfile:
21     # iterate over mcldf object to get voltage magnitudes
22     for row_index, voltages in enumerate(mcldf):
23         # write file header (bus names)
24         if row_index == 0:
25             csvwriter = csv.DictWriter(
26                 csvfile, voltages.keys())
27             csvwriter.writeheader()
28         # write file rows (voltage magnitudes)
29         csvwriter.writerow(voltages)

```

Fig. 2.16 Running a Monte Carlo probabilistic load flow (*prob_lf.py* file)

2.3.3.2 Use of the Probabilistic Load Flow Class

Using the functionality provided by the *MontecarloLoadFlow* class, the *prob_lf.py* script presented in Fig. 2.16 exemplifies how to run a probabilistic load flow. After activating the project and the study case, the *mcldf* generator is created in line 18. Then in line 20, a CSV file is created for storing the load flow results. To obtain the load flow results, the *mcldf* generator is iterated over in line 22, and in every iteration, the results are stored directly in the CSV file to avoid memory availability problems.

2.3.3.3 Set-up and Execution

The set-up and execution procedures of the *prob_lf.py* script are the same as those described in Sect. 2.4.2. In this case, the command *python prob_lf.py* must be entered in the CMD. Once the script is executed, the results are stored in the *res_prob_lf.csv* file.

```

1  from pfsim import PowerFactorySim
2
3
4  FOLDER_NAME = ''
5  PROJECT_NAME = '2A4G'
6  STUDY_CASE_NAME = 'Study Case'
7  MONITORED_VARIABLES = {
8      'G1.ElmSym': ['s:phi', 's:speed', 's:fe'],
9      '*.ElmTerm': ['m:u']
10 }
11
12 # activate project and study case
13 sim = PowerFactorySim(
14     folder_name=FOLDER_NAME,
15     project_name=PROJECT_NAME,
16     study_case_name=STUDY_CASE_NAME)
17 # get all buses in network
18 buses = sim.app.GetCalcRelevantObjects('*.ElmTerm')
19 # create result dictionaries
20 t = {}
21 f = {}
22 for bus in buses:
23     # create short circuit on every bus
24     sim.create_short_circuit(
25         target_name=bus.loc_name+'.ElmTerm',
26         time=2.0,
27         duration=0.15)
28     # prepare RMS simulation
29     sim.prepare_dynamic_sim(
30         monitored_variables=MONITORED_VARIABLES)
31     # run RMS simulation
32     sim.run_dynamic_sim()
33     # get and store generator response
34     t[bus.loc_name], f[bus.loc_name] = \
35         sim.get_dynamic_results('SG1.ElmSym', 's:fe')
36     # delete old short circuit before new one
37     sim.delete_short_circuit()

```

Fig. 2.17 Obtaining the response of a synchronous generator to short circuits at different buses (*rms_sc.py* file)

2.3.4 Example with Dynamic Scenarios: Synchronous Generator Response to Different Short Circuits

The *rms_sc.py* script presented in Fig. 2.17 exemplifies how different dynamic scenarios can be created through events. This script determines the response of one of the synchronous generators in the two-area system from [7] (see Fig. 2.25) to short circuits at different buses by running several scenarios with RMS simulations.

2.3.4.1 The Script

In Fig. 2.17, the *MONITORED_VARIABLES* dictionary indicates to the *prepare_dynamic_sim* method that rotor angle, speed and electrical frequency are to be monitored for one of the generators, and that the voltage magnitude must be monitored at each bus. For each bus in the system, a short circuit of 0.15 s is created and an RMS simulation is run. Every short circuit event is deleted before a new one is created. For simplicity, only the frequency of the generator is retrieved in this example. The results are stored in two dictionaries: *t* (time) and *f* (frequency). The keys of each dictionary are bus names.

2.3.4.2 Set-up and Execution

The set-up and execution procedures of the *rms_sc.py* script are the same as those described in Sect. 2.4.2. In this case, the command *python rms_sc.py* must be entered in the CMD. The results of this simulation are displayed in Fig. 2.21. Section 4.2 expands on how these results can be visualized.

2.4 Data Analysis with Python

Power systems engineering can benefit from automated data analysis just as much as any other branch of engineering. The collection of well-established Python libraries available for these purposes makes the language an increasingly popular choice for data analysis. This section begins by describing the recommended steps to manage Python installations and packages, and it then proceeds to show, by example, how various data analysis tasks can be performed using specialized packages. The examples provide guidelines to analyse the results obtained in previous sections.

2.4.1 Package Management

Managing Python packages can become a source of frustration when there is a need to maintain multiple versions of the Python interpreter on the same computer. Packages compatible with each version of the interpreter must remain identifiable, so the right version of the interpreter and packages is invoked when executing a given project. Virtual environments are a way of solving this problem.

A virtual environment allows creating isolated installations of Python interpreters and their compatible packages. Installations created on different virtual environments can coexist without causing version conflicts and can be in use simultaneously. Even if only one Python installation is present, it is recommendable

```
conda create --name pfpv python=3.4 numpy scipy pandas scikit-lea  
rn matplotlib
```

Fig. 2.18 Creating a virtual environment called *pfpv* for Python 3.4 with the *conda* command

to use virtual environments to avoid future conflicts. The most popular Python tool for creating virtual environments is implemented in the *Virtualenv* package.⁵ However, installing packages with complex dependencies (i.e. that require many other packages in order to function) in a virtual environment can be a nuisance on its own.

Perhaps the easiest way to solve these problems in the realm of data analysis is through Anaconda.⁶ Anaconda is an open source Python distribution that includes a vast collection of Python packages for data analysis as well as a package dependency and environment manager called Conda. Conda allows creating virtual environments for various Python versions and to install compatible packages and their dependencies automatically. A minimal version of Anaconda that only contains Python and Conda, called Miniconda, is also available. Since full Anaconda installers are available only for Python 2.7 and 3.5 by default, but PowerFactory 15.2 requires Python 3.4, it is unnecessary to install the entire package collection included with these Anaconda installers. For this reason, Miniconda is recommended for the examples presented in this chapter. The installation process is straightforward using the Windows installer⁷ and therefore not covered here.

With Miniconda installed, creating a Python 3.4 virtual environment called *pfpv* (as in PowerFactory-Python) with the required packages (NumPy, SciPy, Pandas, scikit-learn and Matplotlib) can be done by introducing the command shown in Fig. 2.18 in the CMD. This prompts Conda to begin fetching and installing packages in the newly created *pfpv* virtual environment.

Activating the environment is the last step required before using it. The activation can be performed with the *activate pfpv* command. All examples in this section must be run from a CMD where the *pfpv* environment is active.

2.4.2 Data Visualization with Matplotlib

Matplotlib [6] is a popular Python data visualization package that can produce a variety of high-quality figures, such as 2D and 3D plots, histograms and scatterplots, among others. Although Matplotlib provides advanced data visualization functionality, producing basic figures is extremely simple.

⁵<https://virtualenv.pypa.io>.

⁶<https://www.continuum.io/>.

⁷<http://conda.pydata.org/miniconda.html>.

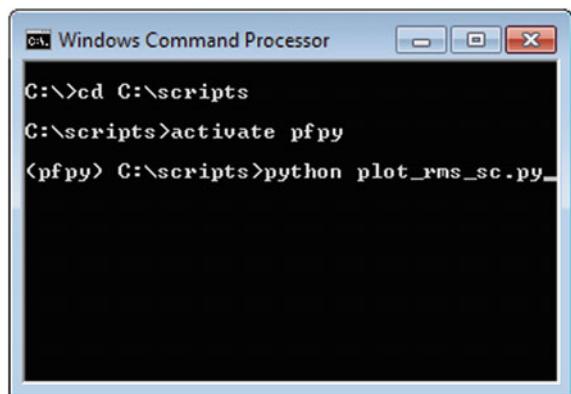
```

1 import matplotlib.pyplot as plt
2
3 # here goes the code from Fig. 17
4
5 :
6
7 # plot generator frequency
8 for bus in buses:
9     plt.plot(
10         t[bus.loc_name],
11         f[bus.loc_name],
12         label=bus.loc_name)
13
14 plt.legend()
15 plt.xlabel('time [s]')
16 plt.ylabel('frequency [p.u.]')
17 plt.show()

```

Fig. 2.19 Plotting the response of the synchronous generator from the example in Sect. 2.3.4 to short circuits at different buses using Matplotlib (*plot_rms_sc.py* file)

Fig. 2.20 Activating the *pfpv* virtual environment and running the *plot_rms_sc.py* script (see Fig. 2.19) located in *C:\scripts* from the Windows command prompt (CMD)



The code snippet in Fig. 2.19 shows how to plot the response of the synchronous generator form the example in Sect. 2.3.4. The set-up and execution of this script are similar to the one described in Sect. 2.4.2, but in this case the *pfpv* virtual environment must be activated before the script is executed, as it is shown in Fig. 2.20. The resulting plot is shown in Fig. 2.21.

2.4.3 Efficient Computing with NumPy and SciPy

Although Python provides a set of versatile high-level data structures, such as lists and dictionaries, these are not well suited for efficient numerical computations.

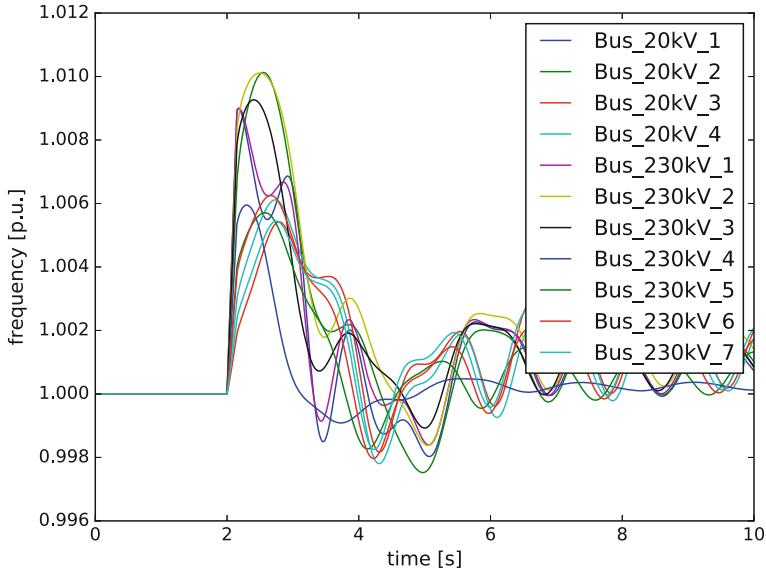


Fig. 2.21 Generator response plot created with the `plot_rms_sc.py` script (Fig. 2.19) using Matplotlib

NumPy [2] is an extension to Python that provides a data structure for efficient computing: the NumPy array. A NumPy array is a multidimensional collection of elements of the same type (i.e. only double precision, only integers), similar to a vector or a matrix. NumPy arrays are efficient mainly due to the way they manage memory and the pre-compiled high-level operations they support. Examples of this are array transposition operations, which only modify the way data is read instead of reorganizing it in memory, and array multiplications, which make it possible to avoid inefficient *for* loops. SciPy [3] is a library that relies on NumPy arrays and provides extended scientific computing functionality, for example, for linear algebra, numerical integration, optimization, signal processing and Fourier analysis.

The code snippet in Fig. 2.22 presents a simple application of SciPy (and therefore of NumPy) to the calculation of the Fourier Transform of a voltage waveform. The voltage waveform is stored in *ula*, which results from the EMT simulation in the example from Sect. 2.2.4. The numerical algorithm used for this purpose is the Fast Fourier Transform (FFT) implemented in the *fft* method. It can be seen that this Python script is neither lengthier nor more cumbersome than a MATLAB script for the same task. The set-up and execution of the *fourier_emt.py* script shown in Fig. 2.22 are the same as the one described in Sect. 2.4.2.

```

1 import matplotlib.pyplot as plt
2 from scipy import fft, fftpack
3
4 # here goes the code from Fig. 7
5
6     :
7
8 # assuming a fixed step size of 0.0001 s
9 step_size = 0.0001
10 # calculate fft of voltage wave
11 ULA = abs(fft(ula))/len(ula)
12 # determine frequency for each harmonic in ULA
13 f = fftpack.fftfreq(ula.size, step_size)
14 # plot voltage waveform and its FFT
15
16     :

```

Fig. 2.22 Calculation of the Fast Fourier Transform of the *ula* voltage waveform obtained from the EMT example in Sect. 2.4 using SciPy and NumPy (*fourier_emt.py* file)

2.4.4 Handling Large Datasets with Pandas

Pandas [4] is a Python library for manipulation and analysis of large datasets. Pandas provides data structures that facilitate working with labelled data, the most widely used being the DataFrame. The DataFrame, inspired by a similarly named data structure from the R language, organizes data in tabular form, comparable to a spreadsheet or an SQL table, where columns and rows are labelled. DataFrames can hold heterogeneous data and support a variety of data rearrangement, selection and description methods.

Figure 2.23 presents the *data_analysis.py* script, which exemplifies some of the basic functionality of Pandas by analysing the results from the probabilistic load flow example introduced in Sect. 2.3.3. The script starts by loading the *res_prob_lf.csv* results file into a DataFrame (line 5). Once the results are in a DataFrame, it is simple to carry out a statistical analysis of the results. Lines 13 and 17 show how to calculate the mean value and standard deviation of the voltage magnitude at each bus, while line 21 takes advantage of the *describe* method to produce a summary of the results. Line 25 demonstrates how to identify buses that at some point exhibit voltages that exceed a certain threshold, in this case, 1.03 p.u. Pandas is well integrated with Matplotlib as well; this is exemplified in line 29, which generates a histogram of the voltages at the *Bus_230kV_4* bus. The set-up and execution of the *data_analysis.py* script from Fig. 2.23 are the same as the one described in Sect. 2.4.2, provided the *prob_lf.py* script from Sect. 2.3.3 is run beforehand, and the *res_prob_lf.csv* results file is available in the same directory.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 # load results from file
5 voltages = pd.DataFrame.from_csv(
6     'res_prob_lf.csv', sep=',', index_col=None)
7
8 # show loaded results
9 print('Voltage magnitudes\n',
10      voltages)
11
12 # calculate mean voltage at each bus
13 print('Mean voltage at each bus\n',
14      voltages.mean(axis=0))
15
16 # calculate voltage standard deviation at each bus
17 print('\nStandard deviation of voltage\n',
18      voltages.std(axis=0))
19
20 # generate statistical summary of voltages
21 print('\nStatistic summary of voltages\n',
22      voltages.describe())
23
24 # find voltages higher than 1.03 p.u.
25 print('\nBuses with voltages higher than 1.03 p.u.\n',
26      (voltages>1.03).any())
27
28 # create histograms for terminal voltages
29 voltages.hist(column='Bus_230kV_4', bins=10)
30 plt.show()

```

Fig. 2.23 Analysing the results from the probabilistic load flow example in Sect. 2.3.3 with Pandas (*data_analysis.py* file)

2.4.5 Machine Learning with Scikit-Learn

The use of machine learning techniques in power systems, and particularly in smart grids, has become commonplace. Among the many applications that have been proposed are prediction of cascading failures [10], load forecasting [11], monitoring of electromechanical dynamics [12] and detection of cyber-attacks [13], just to mention a few. The scikit-learn Python library [5] provides implementations of many machine learning algorithms through a simple interface that is intended for users that require data analysis and machine learning functionality but are not specialists in the field. The scikit-learn library relies on NumPy and SciPy and pays special attention to computational efficiency. Section 2.4.6 presents an example of the use of this library.

2.4.6 Example: Clustering Probabilistic Load Flow Results

To illustrate how scikit-learn can be used in conjunction with PowerFactory, the *cluster.py* script is introduced. This script clusters the voltage magnitudes that result from the probabilistic load flow example from Sect. 2.3.3. The objective of this procedure is to identify groups of adjacent buses that exhibit a similar behaviour. This procedure can also be applied to historical data or windows of data streaming from synchrophasors, and it enables the unsupervised identification of areas that have different control requirements. These areas can then be automatically monitored using a technique such as the one proposed in [14].

Since bus clusters must be formed by adjacent buses, a clustering algorithm that takes the structure of the grid into account is necessary; agglomerative clustering complies with this requirement. Agglomerative clustering is a hierarchical clustering method that recursively merges pairs of clusters according to a measure of similarity. Thus, not only areas can be identified, but also hierarchies of areas, which can be exploited in a hierarchical monitoring and control scheme.

2.4.6.1 The Script

In the *cluster.py* script shown in Fig. 2.24, once the project and study case have been activated, the structure of the grid under analysis is retrieved between lines 29 and 38. By making an analogy between a power grid and a graph, in which buses are the vertices and both lines and transformers are the edges, the structure of the grid can be described by the adjacency matrix of the graph. To cluster the results, the first step is to create an instance of the *AgglomerativeClustering* class, which requires the desired number of clusters and the adjacency matrix to be specified (line 40); later, the results can be clustered as in line 44. The outputs of the clustering are a set of labels that indicate to which cluster each bus belongs. These labels are printed in line 45.

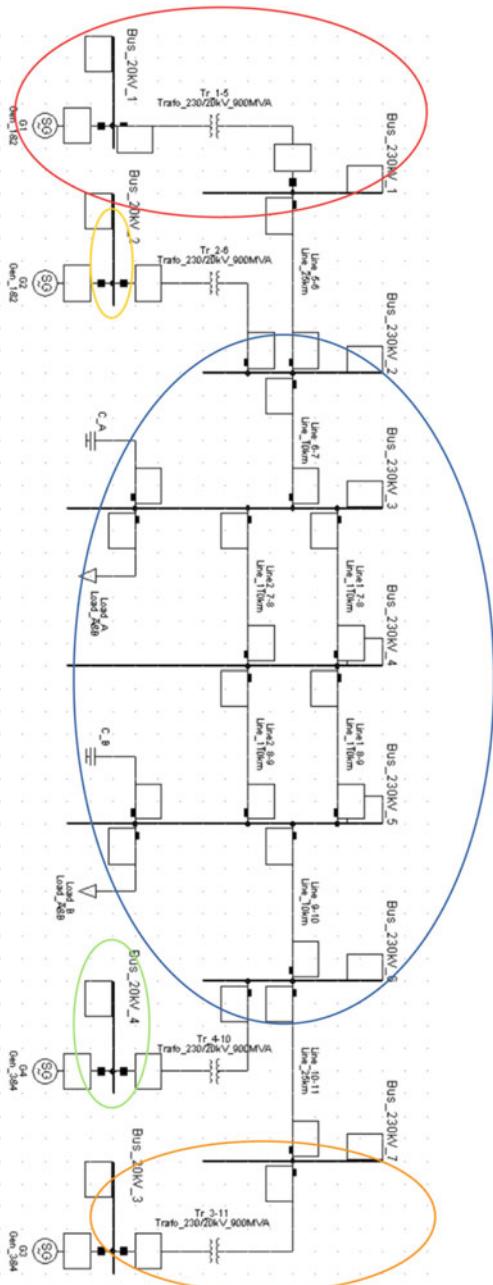
2.4.6.2 Set-up and Execution

Provided that the probabilistic load flow example from Sect. 2.3.3 has already been run, and that the *res_prob_lf.csv* results file is already present in the same directory as the *cluster.py* script, the set-up and execution of this example are the same as the one described in Sect. 2.4.2. The resulting clusters are shown in Fig. 2.25. In addition, Fig. 2.26 shows the bus cluster hierarchy in the form of a dendrogram.

```
1 import numpy as np
2 import pandas as pd
3 from sklearn.cluster import AgglomerativeClustering
4
5 from pfsim import PowerFactorySim
6
7
8 FOLDER_NAME = ''
9 PROJECT_NAME = '2A4G'
10 STUDY_CASE_NAME = 'Study Case'
11
12 # activate project and study case
13 sim = PowerFactorySim(
14     folder_name=FOLDER_NAME,
15     project_name=PROJECT_NAME,
16     study_case_name=STUDY_CASE_NAME)
17 # get elements that connect terminals (lines and
18 # and trafos) into a list
19 lines = sim.app.GetCalcRelevantObjects('*.ElmLne')
20 trafos = sim.app.GetCalcRelevantObjects('*.ElmTr2')
21 links = lines + trafos
22 # load results from file
23 voltages = pd.DataFrame.from_csv(
24     'res_prob_lf.csv', sep=',', index_col=None)
25 bus_names = voltages.columns.values
26 # create zero-filled DataFrame with the structure
27 # of the adjacency matrix
28 adjacency = pd.DataFrame(
29     0, index=bus_names, columns=bus_names)
30 # fill in the adjacency matrix with 1s where a connection
31 # exists
32 for link in links:
33     # get terminals at each extreme of the line or transformer
34     from_bus = link.GetNode(0)[0].loc_name
35     to_bus = link.GetNode(1)[0].loc_name
36     # set corresponding elements in adjacency matrix to 1
37     adjacency.set_value(from_bus, to_bus, 1)
38     adjacency.set_value(to_bus, from_bus, 1)
39 # create an agglomerative clustering object
40 ac = AgglomerativeClustering(
41     n_clusters=5,
42     connectivity=adjacency.values)
43 # cluster the results
44 model = ac.fit(np.transpose(voltages.values))
45 for bus, label in zip(bus_names, ac.labels_):
46     print(bus, label)
```

Fig. 2.24 Clustering voltage magnitudes resulting from the probabilistic load flow example in Sect. 2.3.3 using scikit-learn (*cluster.py* file)

Fig. 2.25 Bus clusters on the two-area system from [7] identified with the *cluster.py* script (Fig. 2.24)



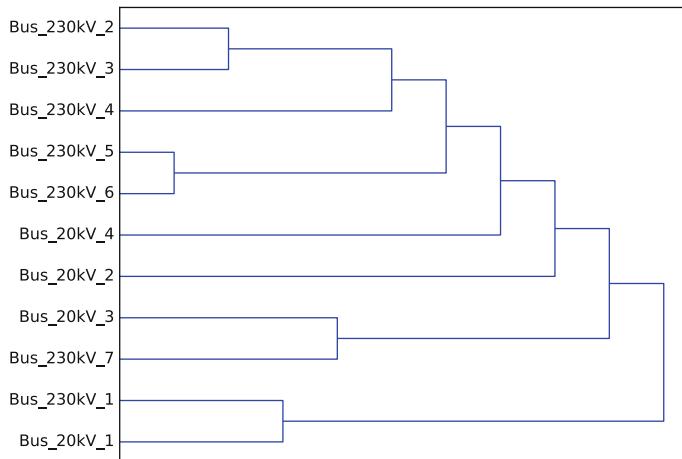


Fig. 2.26 Dendrogram illustrating the bus clusters hierarchy obtained with the *cluster.py* script (Fig. 2.24)

2.5 Conclusions

There is a true need for automating simulation-related tasks, such as scenario generation and processing of large amounts of results. This need is particularly pressing in the case of smart grids. This need can be addressed in diverse ways, one of them being scripting languages. The examples introduced in this chapter show that Python is indeed a useful and versatile tool for supporting and extending the functionality of DIgSILENT PowerFactory, by combining it with well-established Python libraries through the Python API. The limits of what can be achieved from the synergy that is established between PowerFactory and the discussed Python libraries are mainly set by the creativity of the user.

References

1. K.J. Millman, M. Aivazis, Python for scientists and engineers. *Comput. Sci. Eng.* **13**(2), 9–12 (2011)
2. S. van der Walt, S.C. Colbert, G. Varoquaux, The NumPy array: a structure for efficient numerical computation. *Comput. Sci. Eng.* **13**(2), 22–30 (2011)
3. T.E. Oliphant, Python for scientific computing. *Comput. Sci. Eng.* **9**(3), 10–20 (2007)
4. W. McKinney, Data Structures for Statistical Computing in Python, in *Proceedings of the 9th Python in Science Conference*, ed. by S. van der Walt, J. Millman (2010), pp. 51–56
5. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in Python. *J. Mach. Learn. Res.* **12**, 2825–2830 (2011)

6. J.D. Hunter et al., Matplotlib: A 2D graphics environment. *Comput. Sci. Eng.* **9**(3), 90–95 (2007)
7. P. Kundur, N.J. Balu, M.G. Lauby, *Power system stability and control*, vol 7 (McGraw-hill New York, 1994)
8. S. Teimourzadeh, B. Mohammadi-Ivatloo, Probabilistic Power Flow Module for PowerFactory DiGILENT, in *PowerFactory Applications for Power System Analysis*, eds. by F.M. Gonzalez-Longatt, J.L. Rueda (Springer International Publishing, 2014), pp. 61–84
9. B. Slatkin, *Effective Python: 59 Specific Ways to Write Better Python* (Pearson Education, Mar. 2015)
10. S. Gupta, R. Kambli, S. Wagh, F. Kazi, Support-vector-machine-based proactive cascade prediction in smart grid using probabilistic framework. *IEEE Trans. Industr. Electron.* **62**(4), 2478–2486 (2015)
11. P. Zhang, X. Wu, X. Wang, S. Bi, Short-term load forecasting based on big data technologies. *CSEE J. Power Energy Syst.* **1**(3), 59–67 (2015)
12. J. Zhang, C.Y. Chung, Z. Wang, X. Zheng, Instantaneous electromechanical dynamics monitoring in smart transmission grid. *IEEE Trans. Industr. Inf.* **12**(2), 844–852 (2016)
13. M. Ozay, I. Esnaola, F.T.Y. Vural, S.R. Kulkarni, H.V. Poor, Machine learning methods for attack detection in the smart grid. *IEEE Trans. Neural Networks Learn. Sys.* **27**(8), 1773–1786 (2016)
14. R.K. Pandey, S. Kumar, C. Kumar, Development of Cluster Algorithm for Grid Health Monitoring, in *2016 2nd International Conference on Control, Instrumentation, Energy Communication (CIEC)* (Jan. 2016), pp. 377–381

Chapter 3

Smart Network Planning—Pareto Optimal Phase Balancing for LV Networks via Monte-Carlo Simulations



Benoît Bletterie, Roman Bolgaryn and Serdar Kadam

Abstract In this chapter, a user-defined tool to minimise the voltage unbalance caused by unsymmetrical loads and generators is presented. It provides the user with a set of switching actions to improve the power distribution over the three phases. The tool uses the phase connection information provided for example by the automated meter infrastructure and uses a Monte-Carlo-based search to identify the lowest voltage unbalance reachable for a given number of phase-switching actions. By doing this, the Pareto principle can be used, and the user can decide on the necessary number of switching actions and therefore on the efforts needed and justified to improve the situation. The tool, which is implemented via Python scripts, is easily accessible to the user via the user-defined button.

Keywords PowerFactory · Python · Monte-Carlo simulations
Unbalanced load flow · User-defined tool · Pareto principle

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_3) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

B. Bletterie (✉)

Austrian Power Grid AG, Wagramer Straße 19 (IZD-Tower), 1220 Vienna, Austria
e-mail: benoit.bletterie@ait.ac.at

R. Bolgaryn

Fraunhofer IWES, Königstor 59, 34119 Kassel, Germany

S. Kadam

Energy Department, AIT Austrian Institute of Technology GmbH,
Giefinggasse 2, 1210 Vienna, Austria

3.1 Introduction

A significant part of installed photovoltaic generation is connected to low-voltage (LV) networks (e.g. about 70% of the total photovoltaic capacity in Germany is connected to LV networks [1]; the current installed capacity reaching almost 40 GW [2] in Germany by mid-2016). However, the level of information for LV networks is usually very low. In particular, simultaneity factors and the distribution of load and generation over the three phases are in general unknown. With the smart meter rollouts occurring in many countries worldwide, new possibilities are offered. Without accurate information on LV networks, conservative planning rules must be used to ensure that the planning criteria are met for all (or the very large majority) of the networks. With enhanced information on LV networks, distribution systems operators can better determine the available reserves and make a better use of the existing infrastructure. This chapter deals with Phase Balancing for LV networks and tries to answer the question on how to better distribute the power per phase of a large population of unbalanced loads and generators. A tool which has been developed for PowerFactory for this purpose is presented. This issue has gained importance in the recent years, and a non-negligible number of research papers have been published on how to mitigate voltage unbalance created by single-phase PV generators in LV networks [3–7].

3.2 Background—Causes and Consequences of Unsymmetrical Power Flows in LV Networks

Unbalanced conditions in low-voltage networks occur due to the presence of unsymmetrical loads or generators, such as single-phase photovoltaic systems. As a result of the unsymmetrical power flows, the voltages become unbalanced.

The European power quality standard EN 50160 [8] mentions a 2% level for the maximal expectable voltage unbalance (defined as the ratio between negative and positive sequence). One of the main reasons to limit the voltage unbalance (as previously defined) is to avoid or limit the impact on equipment such as three-phase machines (e.g. torque pulsations, overheating) which would lead to a performance degradation.

Besides these pure power quality considerations, the presence of voltage unbalance reduces the available voltage headroom and therefore strongly limits the hosting capacity of low-voltage feeders. Indeed, the voltage rise caused by a single-phase generator is about six times higher than the voltage rise caused by a symmetrical three-phase generator of the same power [3] due to the additional voltage rise on the neutral conductor. Moreover, unsymmetrical power flows might lead to an increase of the current in the neutral conductor leading to higher losses [9] and to a risk of overloading. In rural and residential areas, the hosting capacity is mainly limited by the voltage rise.

Although some concepts have been recently proposed to actively reduce the voltage unbalance by unsymmetrical control of PV inverters [5, 6, 10], trying to solve the problem at its source appears quite natural.

Phase Balancing, which consists of trying to distribute the unsymmetrical power flows over the three phases (by, e.g. switching single-phase photovoltaic generators or EV chargers from one phase to another), can, therefore, be considered as a feasible solution to increase the hosting capacity [3, 4, 9].

3.3 Principle of the Search for a Pareto Optimum Solution via Monte-Carlo Simulations

The basic principle of Phase Balancing tool is to perform a search for the optimal phase configuration (e.g. phase to which single-phase generator or loads is connected) via a Monte-Carlo simulation. For LV feeders with a low number of loads or generators (N), an exhaustive search can be done (number of combinations = 3^N), but for larger (well above 10), an exhaustive search is not feasible, and a Monte-Carlo simulation is proposed. In principle, the number of phase-switching actions required to reach the optimal combination can be as high as the number of generators and loads. Since every phase-switching action supposes workforce efforts, the number of phase assignment changes should be limited, and the concept of Pareto efficiency (also known as the 80–20 rule) is used. Following this concept, the maximal number of phase-switching actions is set (e.g. 5) and the optimal combinations are determined accordingly.

In the proposed tool, the variable to minimise has been chosen to be the voltage spreading. The voltage spreading of a feeder has been defined in [11] as the voltage difference between the highest and the lowest phase for all the nodes (Eq. 1).

$$VS = \frac{\max(U_1, U_2, U_3) - \min(U_1, U_2, U_3)}{U_N} \quad (1)$$

where U_1 , U_2 and U_3 are the phase voltages and U_N is the nominal voltage.

The concept of the search for a Pareto optimum solution via Monte-Carlo simulations is explained with a flowchart in Fig. 3.1.

3.4 Implementation in PowerFactory and Python

3.4.1 General Scheme

The balancing algorithm that is implemented in the aforementioned tool is based on the Monte-Carlo approach. A large number of iterations are performed, so that the load flow calculation delivers a new value of unbalance after a random phase switch

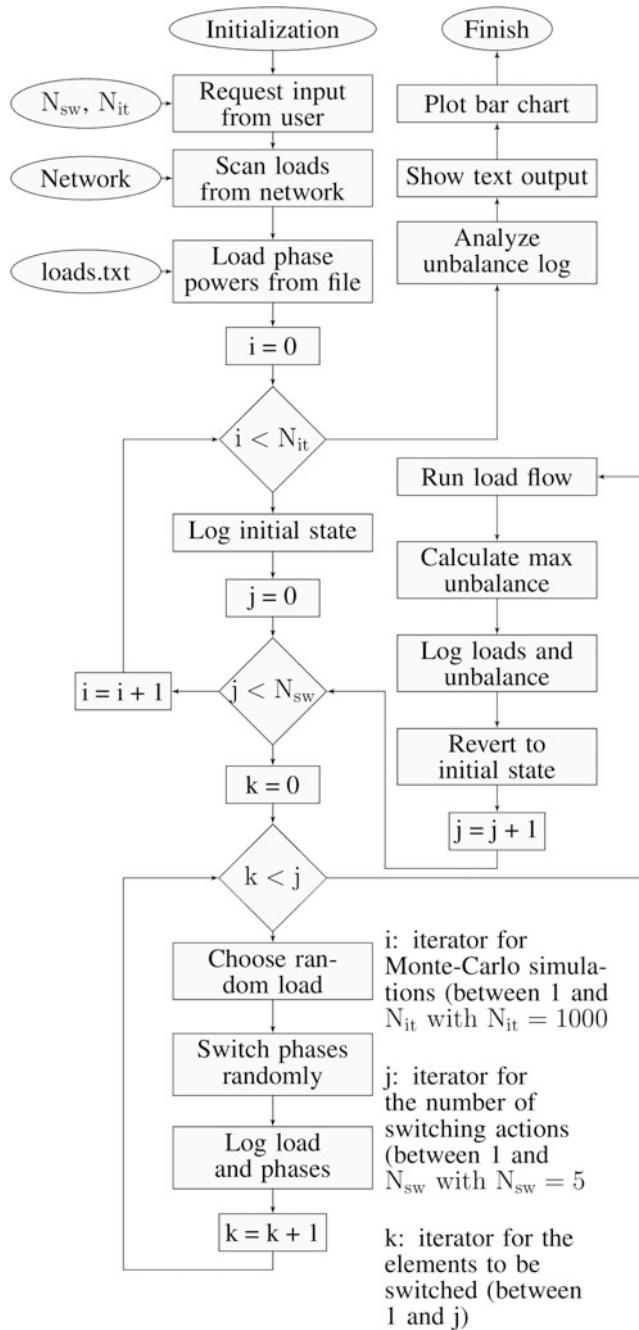


Fig. 3.1 General principle (flowchart)

at every step. The best result (with the lowest unbalance) is selected as the solution (configuration of phase connections) which leads to a lower unbalance.

The phase switches are performed at the buses where unbalanced consumption or generation is present. For example, if there is a power injection on phase A only, then it can be switched to phase B or phase C. The load flow calculation is performed and the maximal voltage unbalance (maximal voltage deviation between the three phases over all the nodes; see Eq. (1)) in the network is calculated.

The network state after every switching is brought back to initial state, and the next iteration starts with a new random set of buses at which the phases are switched. The buses and phases are chosen from a uniform distribution.

The PowerFactory–Python interface is used in order to enable the use of software design opportunities that are present in Python and combine them with the network simulation capabilities of PowerFactory. The version of PowerFactory used for this project is 2017 SP1 and the version of Python is 3.5.2.

The Python script is called from a user-defined tool in the toolbox menu.

Figure 3.2 shows a simplified function call diagram of the tool together with the inputs/outputs.

The network data are collected from the currently activated project. The user provides the path to the text file that contains the phase powers of the loads, and the user defines the number of switching actions to be analysed as well as the number of Monte-Carlo iterations.

The function main_pf.py establishes the connection to the PowerFactory objects and creates the visual interface for the user to input the parameters (Fig. 3.3).

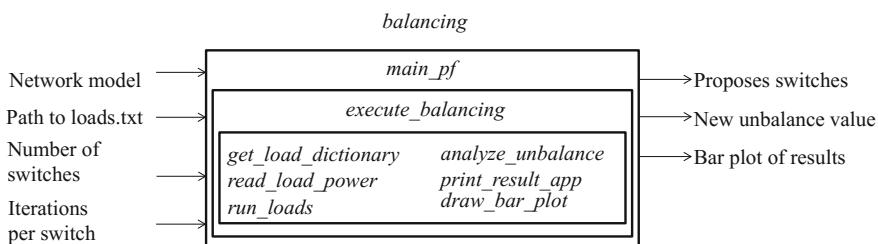


Fig. 3.2 Function call diagram of the tool

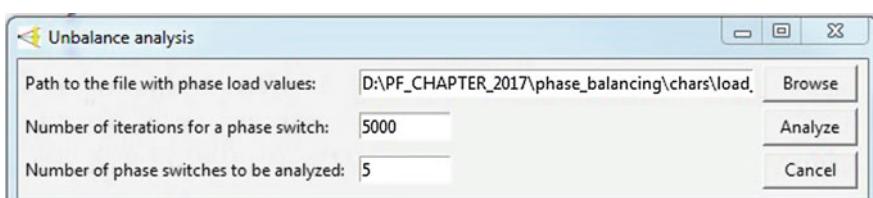


Fig. 3.3 User interface for the input of parameters

The function `execute_balancing.py` is called when the user clicks the button “Analyse”. The main steps of the script are called by this function.

After clicking on the script icon, the interface window of the tool is shown to the user. The window has input parameters that define the path to the file with the load power values, the maximal allowed number of switching actions and the number of Monte-Carlo iterations. The execution of the script starts with calling the function “`main_pf`” when the user clicks the button “Analyse”.

The execution of the script follows the following order:

I. Collecting the load data from the network

The loads of the type `.ElmLod` are all stored in the Python dictionary. They are modelled as unbalanced three-phase loads, and the phase power values define to which phase(s) the load is connected. The phase-switching occurs by rewriting the phase power values.

II. Reading phase powers from a text file

The text file with phase powers for every load with the path that has been provided as input to the script is opened, and the values are input into the loads. These text files can be as previously mentioned automatically created from the database (automated metering infrastructure—AMI or geographic information system—GIS) where this information is available (see Sect. 1.4.2).

III. Monte-Carlo algorithm

The maximal unbalance in the network after every phase switching is recorded and updated into the Python dictionary for every Monte-Carlo iteration.

IV. Analysis of the results

The highest reduction of the unbalance (voltage spreading) for each maximal number of switching actions (e.g. 1, 2, 3, 4, and 5 for a maximal number of 5 switching actions) defines the optimal solution—a new phase configuration.

3.4.2 Data Input (Loads, Phase Information from Smart Meter Data)

The only data necessary for running the script consists of the installed power per phase. This information can be extracted from, for example, an automated metering infrastructure (AMI) database or from a GIS database. In [12], a system for identifying

```
<pss:snapshots>
<pss:data itemID="G23-511-C.01/951-001816" time="2016-04-03T14:33:51" phase="Ly"/>
<pss:data itemID="G23-511-C.01/951-001816" time="2016-04-03T14:42:15" phase="Ly"/>
<pss:data itemID="G23-511-C.01/951-001816" time="2016-04-03T14:33:56" phase="Ly"/>
<pss:data itemID="G23-511-C.01/949-001520" time="2016-04-03T14:33:51" phase="Lx"/>
<pss:data itemID="G23-511-C.01/951-001801" time="2016-04-03T14:33:51" phase="Lz"/>
</pss:snapshots>
```

Fig. 3.4 Sample smart meter information about the phase to which a meter is connected

```

Load11-b 0.0 20.0 0.0
Load12-c 0.0 0.0 20.0
Load21-a -5.0 0.0 0.0
Load22-a -5.0 0.0 0.0
Load23-a -5.0 0.0 0.0
Load24-a -5.0 0.0 0.0
Load25-b 0.0 -5.0 0.0
Load26-c 0.0 0.0 -5.0
Load27-b 0.0 -5.0 0.0
Load28-c 0.0 0.0 -5.0
Load31-a(1)-5.0 0.0 0.0
Load31-a(2)-5.0 0.0 0.0

```

Fig. 3.5 Sample input file (.txt) with the power per phase information

the phase at the meter-clamp at any point of the network has been briefly introduced. A phase recognition function is in the meantime available for most smart meters.

The Phase Balancing tool uses this phase information to get the starting point of the Monte-Carlo search. Figure 3.4 shows as example a .xml file providing the phase information of five meters (phase Lx, Ly or Lz here).

The developed tool reads this information from a text file in which the power values are indicated for each phase. Note that the tool applies the algorithm to the loads and generators (with a negative power) that are modelled as three-phase loads (3PH-YN). The first column of the .txt file specifies the names of the load elements. The following lines specify the power values of the phases A, B and C as shown in Fig. 3.5.

Unknown sources of unbalance data (e.g. unbalanced load within a dwelling) can be ignored (the unbalanced load of a single house can vary strongly during the day). The tool should then only consider systematic unbalance sources such as single-phase (photovoltaic) generators or single-phase charging stations for electric vehicles.

3.4.3 Configuration

The tool is implemented as a user-defined script and can be called directly from the toolbox (Fig. 3.6). In order to enable this feature, the script must be configured in the Administrator mode of PowerFactory.

In order to do so, the following instructions should be followed. For more detailed information on user-defined tools, please refer to the help of [13].

V. Add environment variable

In order to enable Python scripts working with PowerFactory, the path to the file “powerfactory.pyd” in the installation folder of PowerFactory should be added to



Fig. 3.6 PowerFactory toolbox

```
setx PF_PYD "%path%"
```

Fig. 3.7 Definition of the Python path as a system variable (%path% is the path to the file “powerfactory.pyd”)

the system variables as “PF_PYD”. This can be done directly on the command line (see Fig. 3.7).

VI. Import files

The files that need to be imported are contained in the directory “power_factory_objects”. The script file, “run_balancing.pfd”, should be imported to the PowerFactory folder’s “Configuration/Scripts”, and a new icon object (e.g. balancing.IntIcon) should be created in “Configuration/Icons”. After that, the file “Icon.bmp” should be imported to the icon object.

VII. Configure the script path

The path to the script in the “ComPython” object should be configured to lead to the file “main_pf.py” (Fig. 3.8).

VIII. Configure the tool

After that, the settings window “Tool Configuration” can be activated in the menu “TOOLS” (Fig. 3.9). The script file needs to be specified in the left column

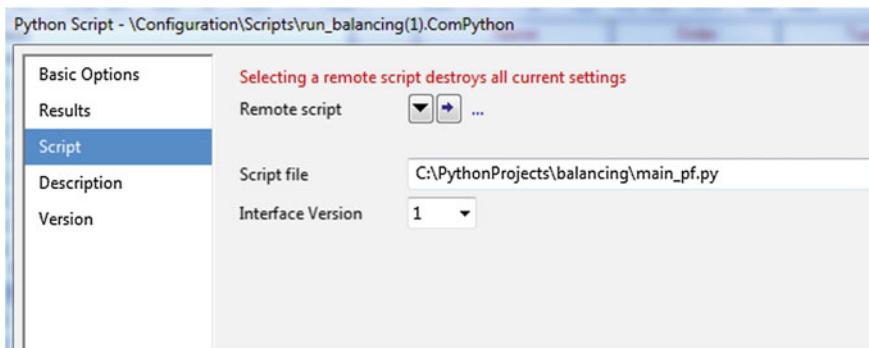


Fig. 3.8 Configuring the script path



Fig. 3.9 Tool configuration

and the icon file in the right column. The script appears in the toolbox of PowerFactory in the category of the user-defined tools.

After the completion of the steps listed above, the tool can be executed by a simple click on the icon.

3.4.4 Python Scripts

This section provides an excerpt of the two main Python functions used in the tool:

- Sample Python script for changing the phase of connection (*run_loads*—see Figs. 3.2 and 3.10)

```
...
#iterate over number of iterations
for i in range(num_loops):
    unbalance[i] = {}
    #iterate over the number of nodes
    for num_nodes in range(max_nodes + 1):
        #switch from 0 to num_nodes
        for j in range(num_nodes):
            #make sure the load hasn't been chosen before and is not symmetrical
            while True:
                load = load_dict[random.choice(list(load_dict.keys()))]
                if load['load_object'].name in new_phases:
                    continue
                elif not load['load_object'].is_symmetrical():
                    break

            #make sure that the random phases are not the same as the initial
            and make sure that the phase switch
            #actually changes the phase powers
            while True:
                phases = random_phases()
                powers = load['load_object'].phase_powers
                if phases != load['load_object'].phases:
                    load['load_object'].phases = phases
                    if load['load_object'].phase_powers != powers:
                        break

            #log the changed load objects and phases
            new_phases[load['load_object'].name] = load['load_object'].phases
    else:
        #after phases are switched, run load flow and do calculations
        ldf.Execute()
        #calculate max unbalance and determine the load at which it occurs
        max_unbalance, critical_load = calc_max_unbalance(load_dict)
        #document everything
        unbalance[i][num_nodes] = {}
        unbalance[i][num_nodes]['unbalance'] = max_unbalance
        unbalance[i][num_nodes]['critical_load'] = critical_load
        unbalance[i][num_nodes]['init'] = dict((k, v) for k, v in init_phases.items() if k in new_phases.keys())
        unbalance[i][num_nodes]['new'] = new_phases
        #return everything to the initial state
        for key, vals in new_phases.items():
            load_dict[key]['load_object'].phases = init_phases[key]
        new_phases = {}

...
```

Fig. 3.10 Sample Python script for changing the phase of connection (excerpt)

```

def calc_max_unbalance(load_dict):
    #find the load in the dict that has the highest unbalance
    critical_load = max(load_dict, key = (lambda load_name:
load_dict[load_name]['load_object'].calc_unbalance()))
    #get the value of unbalance
    unbalance = load_dict[critical_load]['load_object'].calc_unbalance()
    return unbalance, critical_load

def analyse_unbalance(unbalance):
    #get the lowest unbalance for every number of switches
    output = {}
    for key, dict in unbalance.items():
        for subkey, subdict in dict.items():
            if subkey in list(output.keys()):
                if subdict['unbalance'] < output[subkey]['unbalance']:
                    output[subkey]['unbalance'] = subdict['unbalance']
                    output[subkey]['critical_load'] = subdict['critical_load']
                    output[subkey]['init'] = subdict['init']
                    output[subkey]['new'] = subdict['new']
            else:
                output[subkey] = {}
                output[subkey]['init'] = subdict['init']
                output[subkey]['new'] = subdict['new']
                output[subkey]['unbalance'] = subdict['unbalance']
                output[subkey]['critical_load'] = subdict['critical_load']
    return output

```

Fig. 3.11 Sample Python script for the Monte-Carlo simulations (excerpt)

- Sample Python script to analyse the results of the Monte-Carlo simulations (*analyse_unbalance*—see Figs. 3.2 and 3.11).

For more information about the PowerFactory–Python interface, the reader shall refer to [14].

After running the tool, the results stored in the Python variable “*output*” document the phase-switching actions and the corresponding unbalance (voltage spreading) values for every maximal number of switching actions.

3.4.5 Presentation of the Results

The results are presented in the output window of PowerFactory [list of the phase-switching actions and resulting unbalance (voltage spreading)]. Also, the voltage spreading is displayed on a new sheet (*VisPlot* diagram—see example in Sect. 1.5.2).

After completion, the tool restores to the initial state.

3.4.6 Automation (Automatic Execution for a Large Number of Networks)

Through its modular structure, the Python script can be easily integrated into a high-level script which could, for example, analyse on a regular basis a large set of

LV networks to identify heavily unbalanced feeders and prioritise the efforts on the most unbalanced feeders. Example for such a function has been mentioned in [9].

3.4.7 *Extension—Use of Power Snapshots*

Previous research work showed that unbalanced LV networks could be quite complex [3, 11, 15]. In [12], a so-called Power Snapshot Analysis by Meters (PSSA-M) method has been proposed to automatically acquire, store and manage Power Snapshots (fully synchronous measurements of phase voltages and active and reactive powers by every smart meter in a given LV network). The Power Snapshots (stored in a database and exported, e.g. as a *xml* file) could be loaded into PowerFactory (exported) to run the Phase Balancing tool. This would allow optimising the LV networks not only for the installed powers (e.g. installed PV or EV power per phase) but also for taking into account the actual power levels observed per phase during a given time period.

3.5 Application Example

In this chapter, an application example is used to explain the use and results of the Phase Balancing tool. A very simple example has been voluntarily chosen to allow the reader to get familiar with the tool and easily understand the results.

3.5.1 *Test Network*

The network LV network consists of transformer supplying a single feeder with nodes supplied by identical cables (200 m, 50 mm² copper). The network is modelled as a three-phase four-wire network, i.e. with explicit neutral [the neutral is grounded at the LV busbars of the secondary substation and has, in this case, not been grounded at any further node (the grounding resistance is assumed to be large in comparison to the cable impedances)].

All the loads are three-phase unbalanced loads, and as previously mentioned, the generators are modelled as negative loads. The naming of the loads provides the information on which phase they are connected to, and the power per phase is indicated in Fig. 3.5.

The initial distribution of the single-phase loads and generators can be characterised by (Fig. 3.12):

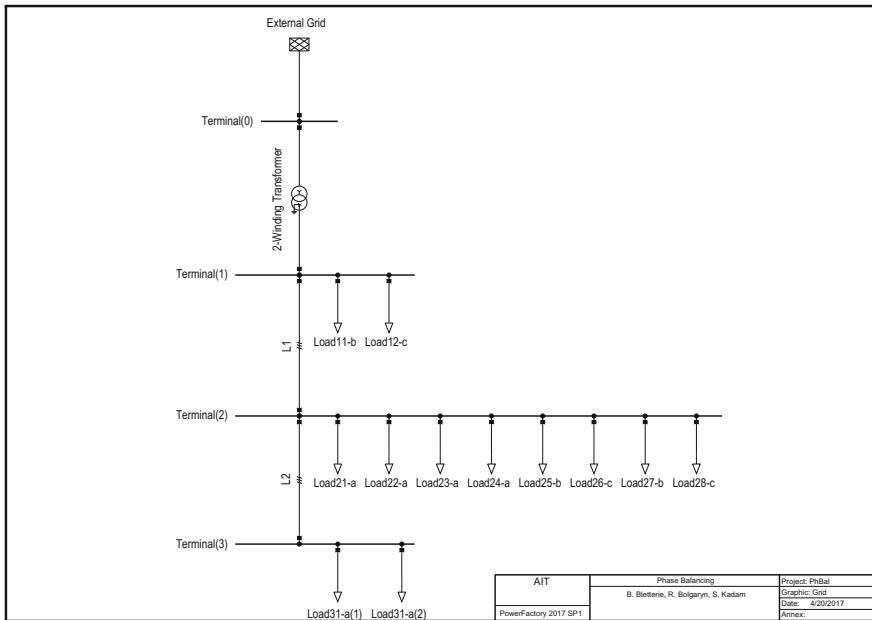


Fig. 3.12 Single-line diagram of the test network

- a moderately unbalanced power flow at the feeder level (net power consumption of 30 kW on phase A and 10 kW on phase B and C).
- a large voltage unbalances (voltage spreading) of almost 11% of the nominal voltage (see feeder voltage diagram on Fig. 3.13).

3.5.2 Results and Comparison with the Built-in Tool

The results obtained after running the Phase Balancing tool are shown in Fig. 3.14 (output window) and Fig. 3.15 (bar plot).

The “init” rows present the information about the initial state. The “new” rows show the phase state after the optimisation. For instance, the maximal voltage spreading could be reduced from 10.9 to 7.1% by switching the phases A to B, B to A and C to B at the load “Load31-a(1)”. Additionally, the load at which the highest unbalance occurs is documented. The initial configuration, before optimisation, has the highest unbalance at the load “Load31-a(2)”. After two switches, “Load24-a” becomes the load with the highest unbalance.

Besides providing the information in the output window, the tool builds a bar plot to show the results graphically (Fig. 3.15).

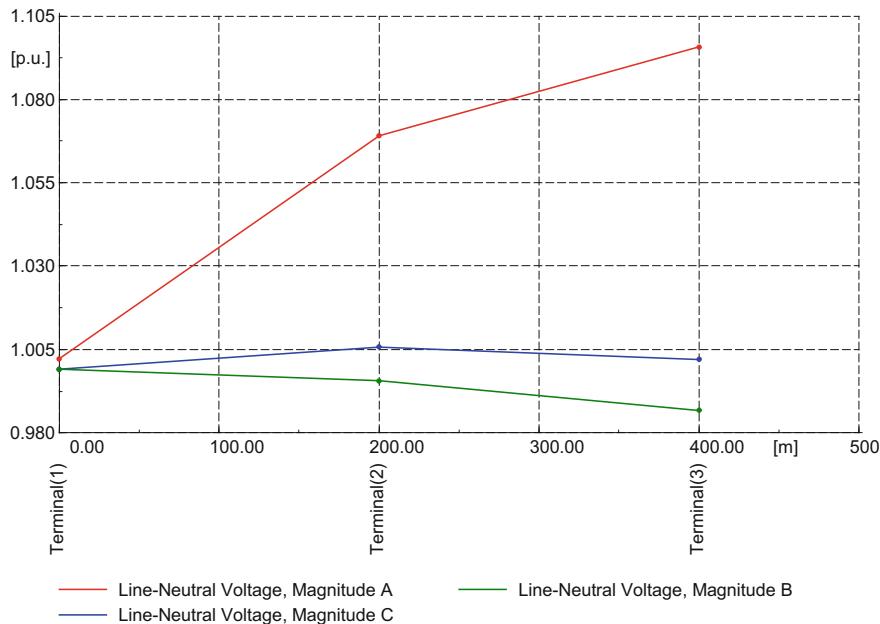


Fig. 3.13 Feeder voltage diagram—initial state

The bar plot shows that in the considered case, using a maximal number of two switching actions leads to a significant unbalance reduction—from 10.887 to 2.030 p.u. Any further switching action will reduce the unbalance by only a marginal amount (following the Pareto principle). The new feeder voltage diagram is shown in Fig. 3.16.

Since a few versions (already in 15.X), PowerFactory offers a *phase balance optimisation* tool in the *distribution network tools* [13]. This tool offers two possible objective functions:

- Objective function 1: minimise average power unbalance.
When selecting this objective function, the tool tries to minimise the overall average power unbalance over all the branch elements.
- Objective function 2: minimise power unbalance at feeding point.
When selecting this objective function, the tool tries to minimise the power flow seen at a feeder point.

PowerFactory proposes simulated annealing as stochastic optimisation method and stores the new phase configuration into a network variation.

Table 3.1 provides a comparison between the following configurations:

- Initial stage (initial network with the initial phase combination; see Sect. 1.5.1)
- Result of the user-defined Phase Balancing tool (for a maximum of two switching actions)

```

Load values have been imported from C:\Program Files\DIgSILENT\PowerFactory 2016
SP3\Tools\phase_balancing\charts\load_values.txt
Number of iterations to be run on up to 5 node(s) is 5000
--- 0 ---
init: {}
new: {}
unbalance: 10.887 % at Load31-a(2)
--- 1 ---
init: {'Load31-a(1)': ('A', 'B', 'C')}
new: {'Load31-a(1)': ('C', 'A', 'B')}
unbalance: 7.054 % at Load31-a(2)
--- 2 ---
init: {'Load31-a(1)': ('A', 'B', 'C'), 'Load31-a(2)': ('A', 'B', 'C')}
new: {'Load31-a(1)': ('C', 'A', 'B'), 'Load31-a(2)': ('C', 'B', 'A')}
unbalance: 2.030 % at Load24-a
--- 3 ---
init: {'Load11-b': ('A', 'B', 'C'), 'Load31-a(1)': ('A', 'B', 'C'), 'Load31-a(2)': ('A', 'B', 'C')}
new: {'Load11-b': ('B', 'A', 'C'), 'Load31-a(1)': ('C', 'A', 'B'), 'Load31-a(2)': ('C', 'B', 'A')}
unbalance: 1.708 % at Load24-a
--- 4 ---
init: {'Load11-b': ('A', 'B', 'C'), 'Load31-a(1)': ('A', 'B', 'C'), 'Load12-c': ('A', 'B', 'C')}
new: {'Load11-b': ('B', 'A', 'C'), 'Load31-a(1)': ('C', 'B', 'A'), 'Load12-c': ('C', 'A', 'B'), 'Load31-a(2)': ('C', 'A', 'B')}
unbalance: 1.517 % at Load24-a
--- 5 ---
init: {'Load11-b': ('A', 'B', 'C'), 'Load27-b': ('A', 'B', 'C'), 'Load22-a': ('A', 'B', 'C'), 'Load23-a': ('A', 'B', 'C'), 'Load31-a(2)': ('A', 'B', 'C')}
new: {'Load11-b': ('C', 'A', 'B'), 'Load27-b': ('A', 'C', 'B'), 'Load22-a': ('B', 'C', 'A'), 'Load23-a': ('C', 'A', 'B'), 'Load31-a(2)': ('B', 'A', 'C')}
unbalance: 1.517 % at Load24-a
elapsed time: 132.323 s

```

Fig. 3.14 Sample result from the output window

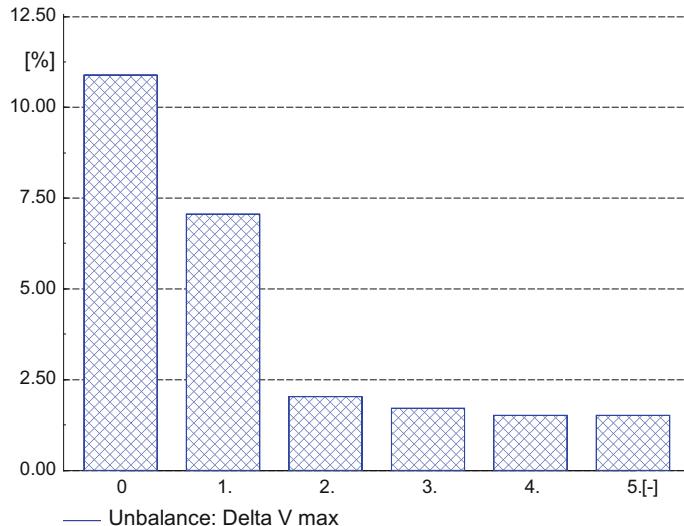


Fig. 3.15 Sample bar plot for a maximal number of switching actions between 0 and 5

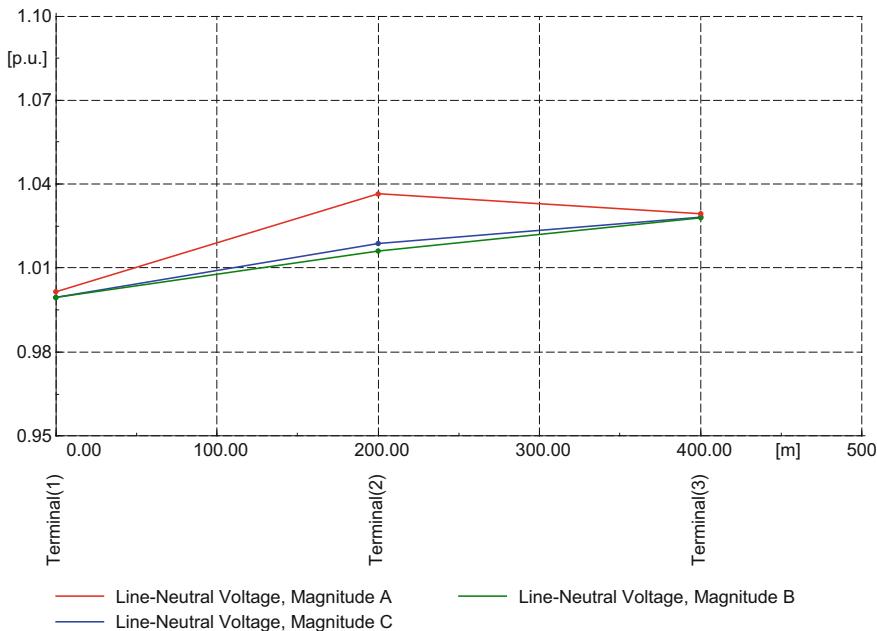


Fig. 3.16 Feeder voltage diagram—optimal phase distribution for a maximal of two switching actions

Table 3.1 Comparison between the user-defined balancing tool and the built-in tools

	Initial state	Phase Balancing tool	Built-in tool obj. function 1	Built-in tool obj. function 2
Number of switching actions	–	2	8	7
Voltage spreading (%)	10.9	2.0	3.3	11.5
VUF (U_2/U_1) (%)	1.5	0.4	0.2	1.3
U_{\min} (p.u.)	0.999	0.999	0.999	1.000
U_{\max} (p.u.)	1.056	1.030	1.035	1.042
Losses (kW)	2.1	1.3	1.3	2.6

- Result of the built-in tool for the first objective function (minimisation of the overall average power unbalance)
- Result of the built-in tool for the second objective function (minimisation of power unbalance at the feeder begin).

This table shows as expected that the built-in tool with the second objective function leads to a situation in which the voltage spreading and the losses are not reduced. This is due to the fact that the objective is only to balance the power

flowing through each phase at the feeder begin, which does not consider at all the feeder properties (impedances) which impact voltage profile and losses.

The built-in tool with the first objective function leads to a significant improvement of the voltage spreading (from 10.9 to 3.3%) and of the network losses (from 2.1 to 1.3 kW). It requires in total eight switching actions to reach this result.

The proposed Phase Balancing tool manages to reach the lowest voltage spreading (decrease from 10.9 to 2.0%) which is logical since it tries to minimise the phase spreading. The decrease of the feeder losses is the same as for the previous case (from 2.1 to 1.3 kW). One of the major advantages is that the script manages to reach this result with only two switching actions (compared to eight for the built-in tool with the first objective function).

The proposed balancing tool allows applying the Pareto principle and determining the best possible phase combination for a given number of switching actions, making it easy for network planners to decide whether the initial situation is worth improvement and for which networks to invest efforts to improve the situation. With this simple tool, which can be fully integrated into a larger tool landscape (including interfaces to other databases), distribution system operators are given the opportunity to customise and automate network planning tools and make use of the new possibilities offered in the smart grids context (e.g. improved access to metering data).

The test network is provided with three network variations to allow a quick comparison between the results:

- Base network: initial configuration (before optimisation)
- “*0.PhaseBalancingTool*”: result of the user-defined Phase Balancing tool presented in this chapter.
- “*1.BalanceOpt@Feeder_1*”: result of the built-in tool with the first objective function.
- “*2.BalanceOpt@Feeder_2*”: result of the built-in tool with the second objective function.

3.6 Conclusions

The proposed user-defined tool allows determining the optimal allocation of unbalanced loads and generators over the three phases in any LV network. Through the use of Monte-Carlo simulations, the best phase combination can be determined for different numbers of maximal switching actions. In order to run, the tool only needs the information of the installed power per phase for each unbalanced load or generator (unknown sources of unbalance should be ignored since they are, in general, subject to variations).

The tool which is implemented via Python scripts is easily accessible to the user via the user-defined button. This way, it can be used by anyone familiar with basic

PowerFactory functions without needing any specific programming knowledge. The script has been programmed in a way that it can be easily distributed.

The results of the Phase Balancing tool are presented in a visual way which allows network planners to decide on whether the initial situation is worth improvement and how much effort should be devoted to it by following the Pareto principle.

As a result, heavily unbalanced feeders can be identified and prioritised in order to flatten the voltage profile and to reduce the voltage unbalance (voltage spreading) [3, 9].

The proposed tool can, therefore, be one of the building blocks within a larger smart planning tool which could make use of the available information from different databases (GIS, AMI) in order to get a better knowledge of the real status of LV networks. Due to the fact that a large share of the distributed energy resources is connected to the LV level (roughly 70% in Germany [1]), advanced planning tools can be allowed to identify the available reserves (e.g. hosting capacity), which in turn allows to make a better use of the existing infrastructure.

3.7 Outlook

As an outlook, the tool could be enhanced by allowing to specify the possible switching actions (e.g. allow permutation at loads and generators only (currently implemented) or also at branch elements—maintaining, of course, the phase order to keep the rotation direction). Moreover, a stochastic optimisation could be used instead of the Monte-Carlo search to improve the performance although the performance is good enough.

Finally, the tool can be integrated into a comprehensive tool landscape to allow a fully automated evaluation of a large set of LV networks.

References

1. Stetz, *Autonomous Voltage Control Strategies in Distribution Grids with Photovoltaic Systems: Technical and Economic Assessment* (Kassel University press GmbH, 2014)
2. Installed Power in Germany|Energy Charts. [Online], Available: https://www.energy-charts.de/power_inst.htm. Accessed 10 Aug 2016
3. B. Bletterie, S. Kadam, R. Pitz, A. Abart, Optimisation of LV networks with high photovoltaic penetration—Balancing the grid with smart meters, in *Proceedings of the PowerTech 2013 IEEE Grenoble*, 2013, pp. 1–6
4. S. Weckx, C. GonzalezDeMiguel, P. Vingerhoets, J. Driesen, Phase switching and phase balancing to cope with a massive photovoltaic penetration, in *Proceedings of the Electricity Distribution (CIRED 2013), 22nd International Conference and Exhibition on*, 2013, pp. 1–4
5. S. Weckx, J. Driesen, Load balancing with EV chargers and PV inverters in unbalanced distribution grids. *IEEE Trans. Sustain. Energy* **6**(2), 635–643 (2015)

6. R.D. Lazar, A. Constantin, Voltage balancing in LV residential networks by means of three phase PV inverters, in *Proceedings of the 27th European Photovoltaic Solar Energy Conference and Exhibition*, Frankfurt, 2012, pp. 4068–4071
7. R. Bolgareyn, Compensation of voltage unbalance in LV networks to enhance the hosting capacity, Sep 2015
8. CENELEC, Voltage characteristics of electricity supplied by public electricity networks, 01 Mar 2011
9. G. Roupioz, X. Robe, F. Gorgette, First use of smart grid data in distribution network planning, in *22nd International Conference and Exhibition on Electricity Distribution (CIRED 2013)*, 2013, pp. 1–4
10. S. Weckx, C. Gonzalez, J. Driesen, Reducing grid losses and voltage unbalance with PV inverters, in *Proceedings of the IEEE PESGM*, Washington, 2014
11. B. Bletterie, A. Abart, S. Kadam, D. Burnier, M. Stifter, H. Brunner, Characterising LV networks on the basis of smart meter data and accurate network models, in *Proceedings of the Integration of Renewables into the Distribution Grid, CIRED 2012 Workshop*, 2012, pp. 1–4
12. A. Abart, D. Burnier, B. Bletterie, M. Stifter, H. Brunner, A. Lugmaier, A. Schenk, Power Snapshot Analysis: A new method for analyzing low voltage grids using a smart metering system, in *Proceedings of the Electricity Distribution (CIRED 2011), 21st International Conference and Exhibition on*, Frankfurt, 2011
13. DIgSILENT, DIgSILENT PowerFactory 2016—User Manual, June 2016
14. DIgSILENT, DIgSILENT PowerFactory. Technical Reference Documentation—Python Function Reference, 23 June 2016
15. B. Bletterie, A. Gorsek, A. Abart, M. Heidl, Understanding the effects of unsymmetrical infeed on the voltage rise for the design of suitable voltage control algorithms with PV inverters, in *Proceedings of the 26th European Photovoltaic Solar Energy Conference and Exhibition*, Hamburg, 2011, pp. 4469–4478

Chapter 4

Co-simulation with DIgSILENT

PowerFactory and MATLAB: Optimal Integration of Plug-in Electric Vehicles in Distribution Networks



J. Garcia-Villalobos, I. Zamora, M. Marinelli, P. Eguia and J. I. San Martin

Abstract Smart grid concept is gaining more and more importance in electric power systems. In near term, electric grids will be more intelligent, interconnected and decentralised. Dealing with a significant number of distributed resources in a smart way frequently requires the use of optimal control techniques, which find the best solution according to a defined objective function. Taking into account all these aspects, the simulation of these types of problems is characterised by having a great number of controlled resources and the use of advanced control techniques. In this context, DIgSILENT PowerFactory provides useful tools to simulate complex systems. On the one hand, the DIgSILENT Programming Language (DPL) can be used for multiple purposes such as automation of simulations, automatic generation of simulation scenarios, analysis of results. On the other hand, the DIgSILENT Simulation Language (DSL) and the digexfun interface allow the implementation of advanced control techniques. Using the digexfun interface, DIgSILENT PowerFactory can send and receive data from other mathematical software APIs such as MATLAB. This chapter presents a co-simulation framework developed to test optimal control methods for root mean square (RMS) simulations on

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_4) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

J. Garcia-Villalobos (✉) · I. Zamora · P. Eguia · J. I. San Martin

Department of Electrical Engineering, University of the Basque Country
(UPV/EHU), Bilbao, Spain
e-mail: javier.garcia@ehu.eus

M. Marinelli

Department of Electrical Engineering, Center for Electric Power and Energy,
Technical University of Denmark (DTU), Roskilde, Denmark

DIgSILENT PowerFactory. As an example, the implementation of a smart charging control for plug-in electric vehicles in electric distribution networks is explained.

Keywords Co-simulation · Digexfun interface · Optimal control
DPL · Plug-in electric vehicles

4.1 Introduction

DIgSILENT PowerFactory has become one of the most versatile software tools to analyse electric power systems. Nowadays, this property is essential to model and simulate smart grid-based solutions, especially if there is significant number of intelligently distributed resources such as smart inverters, distributed storage, plug-in electric vehicles. As the number of controlled devices increases, the necessity of implementing coordination agents in different control levels becomes essential. Furthermore, researchers must deal with long and repetitive tasks, such as modelling and place each distributed device along the network under analysis.

In this context, the *DIgSILENT Programming Language* (DPL) serves the purpose of offering an interface for automating tasks [1]. These tasks can also include control of simulations, defining or changing control parameters, modifying network architecture, managing and visualising results, etc. However, one of the drawbacks of DPL is that it cannot be used for changing a parameter while a RMS simulation is being carried out. Also, it is not suited for managing a large amount of data because of the lack of data structures integrated into DPL.

In order to deal with this problem, the DIgSILENT Simulation Language DSL can be used with the digexfun interface. This interface allows for continuous-time and discrete-time models to be linked into RMS or EMT simulations. Through the digexfun interface, user-defined DSL functions can be programmed. This way, PowerFactory may call the user-defined DSL function multiple times per each time step. The *digexfun.dll* file is stored in the PowerFactory installation directory and cannot be modified when PowerFactory program is opened.

Each time PowerFactory starts, the *digexfun.dll* file, which contains the user-defined functions, is read. There is a folder named *digexfun*, located inside the PowerFactory installation directory, which contains an example and necessary source code files for creating a *digexfun.dll* file. The Microsoft Visual Studio software is recommended to compile and generate this file.

In this chapter, the DPL scripting for automatically generate and place distributed devices, such as electric vehicles (EV), is introduced in Sect. 4.2. In general, this work is carried out manually, but for a large number of distributed devices and cases, it is necessary to develop a solution based on DPL scripting so as to avoid time losses and errors. Then, the required co-simulation framework which allows complex control capabilities in RMS simulations is presented in Sect. 4.3. A new user function is developed, and the interchange of data between this new

user function and MATLAB software is explained. In Sect. 4.4, an example project is developed. In this example, an optimised charging control algorithm for plug-in electric vehicles is introduced and implemented by using the digefun interface. Finally, the chapter conclusions are summarised in Sect. 4.5.

4.2 Generation of Simulation Scenarios

Generation of different simulation scenarios requires a huge amount of time, especially when dealing with different penetration levels of distributed resources. Three main tasks should be accomplished to build a simulation scenario: creating models of distributed resources, assigning parameters to them and, finally, placing them in the network. In this chapter, the considered distributed resources are electric vehicles.

The first task includes modelling an electric vehicle, which is used as a base model and creates a copy of it as many times as necessary, according to researcher's criteria. The number of electric vehicles connected to a specific network depends on different global variables, such as a number of houses in the network, the location of the network and penetration level. The second task consists in assigning different parameters to the electric vehicles located in the network. These values may depend on stochastic models which can be obtained from data collected in national household travel surveys [2] or EV pilot projects. In the last task, electric vehicles must be placed in the different nodes of the network so, previously, the characteristics of the network must be extracted. Apart from these tasks, it is always interesting to make the DPL script as versatile as possible, especially if more than one network model will be considered in the analysis. Figure 4.1 shows the required tasks for generation of a simulation scenario, being N the total number of electric vehicles to be generated and placed in the network and n the number of the last electric vehicle that has been placed in the network.

4.2.1 Defining the Model Characteristics

Several previous steps must be performed prior to generate a simulation case with an important number of electric vehicles. Obviously, the first step is modelling electric vehicles which will be used as base models. Models should be developed using a composite model in DSL. This way, key parameters such as active or reactive power set point can be modified during simulation time through user-defined functions. Design parameters of the base models, which are common to all instances of them, should be defined accordingly. The rest of design parameters will be defined by the DPL script.

Three different types of electric vehicles have been considered. Two of them are connected to the network through a single-phase charger and the other one through a three-phase charger. Table 4.1 shows the characteristics of the different models of EVs considered in this work and the probability of appearance in the network.

Fig. 4.1 Flowchart for generation of a simulation scenario

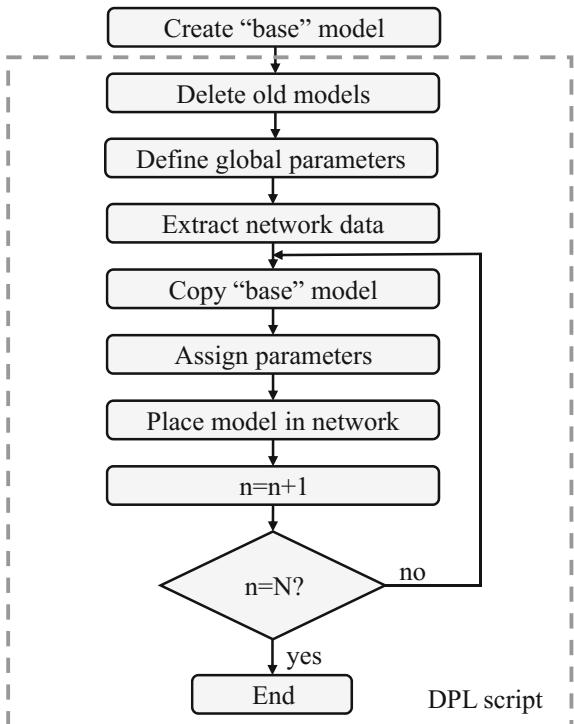


Table 4.1 Characteristics of modelled plug-in electric vehicles

	Type	Capacity (kWh)	Fuel economy (Wh/km)	Efficiency (%)	Charger P. (kW)	Prob. app. (%)
Nissan leaf	BEV	24	212	86.4	3.7 (1ph.)	53
Chevy volt	PHEV	10.3 ^a	239 ^b	83.7	3.7 (1ph.)	27
Tesla S	BEV	60	237	86.4	11 (3ph.)	20

^aUsable battery capacity

^bIn electric mode

Once DSL models have been made and their proper operation has been checked, the composite models must be saved inside the developed DPL scripts, in the Data Manager of PowerFactory. This way, the models and their components, provided that they are located inside the DSL composite model, are local variables of the DPL script. Therefore, the copy processes of the base models are simplified.

Another interesting previous step is deleting the old composite models of EVs produced by previous simulations. Deleting them manually could be a tedious and

```

! --- ClearOld DPL ---
object O;                                     ! variables for objects
set S;                                         ! variables for sets

! Delete composite models of old plug-in electric vehicles
S = AllRelevant("*_EV.ElmComp");               ! get all instances of EVs
for(O=S.First();O;O=S.Next()){                  ! iterate over all objects in the set
    Delete(O);                                 ! delete object
}

!Deleting old folders
S = AllRelevant('*._ElmNet');
O=S.FirstFilt('Network._ElmNet');
S=O.GetContents('*_EV_*._IntFolder');
for (O=S.First();O;O=S.Next()){
    Delete(O);
}

! Delete old cubicles
S = AllRelevant('Cubicle_ev*.Stacubic');       ! get all cubicles of EVs
for(O=S.First();O;O=S.Next()){                  ! iterate over all objects in the set
    Delete(O);                                 ! delete object
}

```

Fig. 4.2 DPL script for deleting old models of EVs

error-prone work. Figure 4.2 shows a short script to delete those old components of previous simulations. Cubicles, which are elements to define the connection between components and terminals, should also be deleted.

4.2.2 Defining Total Number of EVs in the Network

After creating the DSL models and saving them inside the correspondent DPL scripts, the next step is to define the total number of EVs that will be connected to the network. This total number depends mainly on three parameters:

- Number of houses in the network, which is extracted knowing the names assigned to the terminals where houses are connected. In this case, names of the terminals must be previously named following a common pattern. For example, adding “_HOME” to the terminal name.
- Type of areas (rural or urban). The number of EVs follows a normal distribution, which is different for rural or urban areas.
- Penetration level of EVs depending on the scenario.

The DPL has a limited capability to manage stochastic distributions. There is an instruction called *fRand(mode,p1,p2)* to provide double values, which follows a

Table 4.2 Options of fRand instruction

Mode	Type of distribution	Parameter p1	Parameter p2
1	Uniform	Min	Max
2	Normal	Mean	Standard deviation
3	Weibull	Shape	Scale

```

! --- CaseCreator DPL ---
! Declaration of variables
int houses,rural_urban,penetration_level;           ! Integer variables
int type_ev,num_ev,n_total_ev;                      ! Integer variables
double n_ev;                                         ! Double variables
object o_grid,o_term,o_folder;                      ! Object variables
set s_grid,s_term;                                  ! Set variables

! Input data to calculate the total number of EVs
input(rural_urban,'Enter type of area. Type 0 for rural or 1 for urban');
input(penetration_level,'Enter penetration level of EVs in %');

! Generate a folder to store composite models of EVs
s_grid = AllRelevant('Network.ElmNet');
if(s_grid=NULL){
  Error('There is no valid grid name in the network');
}
o_grid = s_grid.First();
o_folder = o_grid.CreateObject('IntFolder','Case_EV_',penetration_level);

! Extract all terminals where homes are connected
s_term = AllRelevant('*_HOME*.ElmTerm');           ! get all instances of terminals
if(s_term=NULL){
  Error('There is no valid terminal name in the network');
}

! Initiate random seed. Must be changed in each execution
SetRandSeed(2);

! Iterate over all objects in the set of terminals. Set number of EVs/home
n_ev = 0;
for(o_term=s_term.First();o_term;o_term=s_term.Next()){
  if(rural_urban=0){
    n_ev += fRand(1,2.351,1.2446);                ! Normal distribution for rural areas
  }
  else if (rural_urban=1){
    n_ev += fRand(1,1.928,1.1036);                ! Normal distribution for urban areas
  }
}

! Total number of EVs
n_total_ev = round(n_ev*penetration_level/100);

! Iterate to call different subscripts and generate EVs
for(num_ev=1;num_ev<n_total_ev;num_ev+=1){
  type_ev = Random();
  if (type_ev<0.53){
    nissan_leaf.Execute(o_folder,s_term,trimester,type_day,num_ev);  ! Subscript
  }
  else if(type_ev>0.8){
    tesla_s.Execute(o_folder,s_term,trimester,type_day,num_ev);      ! Subscript
  }
  else{
    chevy_volt.Execute(o_folder,s_term,trimester,type_day,num_ev);   ! Subscript
  }
}

```

Fig. 4.3 DPL script for defining global simulation parameters

distribution function (uniform, normal or Weibull), as shown in Table 4.2. For other types of distributions, the inverse cumulative distribution function, also known as quantile function, and the *Random* instruction for number generation between 0 and 1 should be used.

Figure 4.3 shows the DPL script used for defining the total number of EVs placed in the network. First, a number of data inputs from the researcher are required. Then, a folder is generated inside the network data to store the new composite models of EVs. After that, home terminals are extracted from the grid and total number of EVs is calculated. Additionally, this DPL script is also used for calling the subscripts which are in charge of generating the different types of EVs (Nissan Leaf, Chevrolet Volt and Tesla Model S).

4.2.3 *Creating and Placing Plug-in Electric Vehicles in the Network*

The following task is to create a copy of the selected EV model type, assign the required characteristics and connect it to the network. Apart from model characteristics, it is necessary to develop a so-called driving and charging model. Driving and charging behaviour of EV drivers are key factors to evaluate the impacts on the electric grid. Questions such as how many kilometres per day are done, how many times per day the EVs are plugged into the network and where and when users plug or unplug their EVs have to be analysed. Furthermore, the answer to these questions depends on the type of areas (rural or urban), region or country, type of day (weekday or weekend day) and other random factors such as weather, sport events. As a consequence, it is difficult to obtain an accurate model of driving and charging behaviour. In this research work, a simplified driving and charging behaviour model has been used:

- Arrival and departure time of each EV. They are defined following a normal distribution. It is assumed that EV drivers connect their vehicles to the network as soon as they arrive at home.
- Initial SOC. It depends on the total number of kilometres travelled by the EV driver in the day.
- Required final SOC. All EVs are charged until they achieve 100% of SOC which is set by default. EV users could set another final SOC.

Figure 4.4 shows a simplified version of the DPL script, which has been used to generate and place a composite model based on Tesla Model S. As part of the composite model, the static generator (*ElmGenstat*) is an easy-to-use model to represent any device connected to the network through a full converter. Modifying d-axis and q-axis current references (*id_ref* and *iq_ref*) active and reactive power set point of the static generator can be modified. Apart from the static generator, two DSL blocks have been developed to represent the battery and the controller of the EV. The user-defined function is called from the last one.

After making a copy of the composite model, its components are extracted in order to change their properties. Then, a random selection of a terminal is carried

```

! --- Tesla Model S subscript ---
int n_term,term;                                ! Integer variables
double aT,dT,distance,energy_per_km;             ! Double variables
double battery_capacity,initial_soc;              ! Double variables
object o_term,o_cubic,o_comp,o_gen,o_pq;         ! Object variables
object o_dsl_controller,o_dsl_battery,o_V;        ! Object variables
string st_ev_name;                             ! Set variable
set s_comp;
! Tesla Model S characteristics
battery_capacity = 60;                         ! kWh
energy_per_km = 0.237;                          ! kWh/km

! Create a copy of the composite model in the folder
o_comp = o_folder.AddCopy(CompositeModel_Tesla,'',num_ev);
st_ev_name = sprintf('%0.3d_TESLA_EV',num_ev);      ! Change name of composite model
o_comp:loc_name = st_ev_name;

! Extract components of the composite model
s_comp = o_comp.GetContents('*',1);
o_gen = s_comp.FirstFilt('.ElmGenstat');
o_pq = s_comp. FirstFilt('.StaPqmea');
o_dsl_controller = s_comp.FirstFilt('Controller.Elmdsl');
o_dsl_battery = s_comp.FirstFilt('Battery.Elmdsl');

! Random selection of a terminal
n_term = s_term.Count();
term = Random(0,n_term);
term = trunc(term);
o_term = s_term.Obj(term);

! Generate a copy of the cubicle instance and configure it as three-phase
o_cubic = o_term.AddCopy(CubicEV,'Cubicle_ev_',num_ev);
o_gen:phtech = 0;                                ! Configure as three-phase cubicle
o_cubic:obj_id = o_gen;                          ! Connect it to the static generator
o_pq:cubic = o_cubic;                           ! Connect the power measurement device
o_V:pbusbar = o_cubic;                          ! Connect the voltage measurement device

! Generate arrival time and departure time
aT = fRand(1,25200,5400);
dT = fRand(1,68400,3600);

! Calculate initial SOC
distance = fRand(1,32,21);
initial_soc = (batt_cap-distance*energy_per_km)/batt_cap;
while(initial_soc<0 .or. initial_soc>1){
    distance = fRand(1,32,21);
    initial_soc = (batt_cap-distance*energy_per_km)/batt_cap;
}
! Assigning variables to the DSL blocks
o_dsl_controller:departureTime = dT;
o_dsl_controller:arrivalTime = aT;
o_dsl_controller:final_soc = 1;
o_dsl_controller:ev_id = num_ev;
o_dsl_battery:SOci = initial_soc;

```

Fig. 4.4 DPL script for creating and placing a Tesla Model S in the network

out and the cubicle is copied and configured. Finally, the different parameters are calculated and the DSL blocks are updated with the new values.

After running the script of Fig. 4.3, which calls subscripts such as the one shown in Fig. 4.4 and others, all EVs are placed. Afterwards, initial conditions of the RMS simulation are set up.

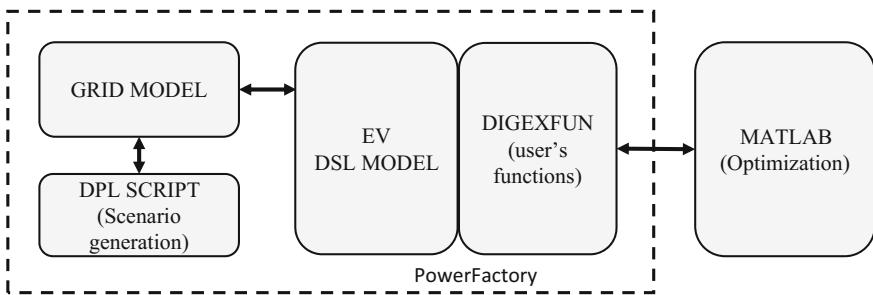


Fig. 4.5 Co-simulation framework

4.3 Co-simulation Framework

The presented co-simulation framework is based on *digexfun* interface, as shown in Fig. 4.5. In order to develop the new user's functions, the *digexfun.dll* file must be modified. A program such as Microsoft Visual Studio allows programming in C++, compiling and creating new *dll* files. As said before, inside PowerFactory installation directory, a folder named *digexfun* contains the necessary files to start developing the new user's functions. The use of *digexfun* interface offers several and important advantages:

- Allows modifying parameters during RMS/EMT simulation time. This is especially advantageous when dealing with a considerable number of variables to be changed at each time step of the RMS/EMT simulation.
- Allows using the potential of C++ as an object-oriented programming language. That is, elements such as classes and objects can be used which is particularly useful for managing distributed resources.
- Allows interfacing with C++ APIs of other software tools, such as MATLAB. This way, complex control functions can be developed and applied during RMS simulations. Additionally, MATLAB offers methods to save results obtained from PowerFactory simulations.

4.3.1 Interface Preparation

Several steps have to be developed before start programming a new user's function. The developed co-simulation framework is based on using Microsoft Visual Studio 2010 (VS2010) and the 32 bits version of PowerFactory. However, newer versions of Visual Studio can be used. The following steps, in VS2010, are necessary to start developing a new user's function properly:

- Solution platform for VS2010 has to be defined as Win32. Using the 64 bits version of PowerFactory and the Win32 solution platform will lead to severe errors.
- Open in VS2010 the file *digexfun.dsp* (in *digexfun* folder) which contains an example project. As *dsp* extension is old, VS2010 will ask to convert it to *vcproj* and open the project. After this first run, *vcproj* file should be used to open the *digexfun* project.
- Open *userfun.cpp* file from solution explorer panel of VS2010. This is the file where user-defined functions are programmed.

4.3.2 Defining New User's Functions Through Digexfun Interface

Developing a new user's function requires a number of steps. The first one is to register the name of the user's function (*cnam*) and the number of input variables (*iargc*) passed as parameters, from PowerFactory to the user-defined function. Figure 4.6 shows an example of the *RegisterFunctions*, with two new user-defined functions (*UncontrolledCharging* and *SmartCharging*).

Both user-defined functions are in charge of setting the required *id_ref* at each simulation time step, taking into account the parameters of each EV. In this regard, the *UncontrolledCharging* function is designed to charge EVs as soon as they are connected to the network. In contrast, the *SmartCharging* function is developed to optimally charge EVs to reduce network impacts. Note that as new user-defined functions are introduced, the variable *ifun* must be numbered consecutively.

The second step consists in adding the function name and the number to the export file, named *digexfun.def* (Fig. 4.7), which is part of the Visual Studio project.

```
// RegisterFunctions C++ function in userfun.cpp

int __stdcall RegisterFunctions(int ifun, char* cnam, int* iargc)
{ // register userdefined functions number ifun:
  // return name (cnam[50]) and number of arguments (iargc)
  // return != 0, if ifun exceeds the number of available functions
  if (!cnam || !iargc) return 1;
  LoadDigsilLibrary();
  if (ifun == 0) {
    strcpy(cnam, "UncontrolledCharging");
    *iargc = 9;
  }else if (ifun == 1) {
    strcpy(cnam, "SmartCharging");
    *iargc = 9;
  }else { // for unknown function
    return 1;
  }
  return 0;
}
```

Fig. 4.6 The *RegisterFunctions* function in *userfun.cpp*

```

LIBRARY    digexfun
EXPORTS
    RegisterFunctions @1
    Init @2
    UncontrolledCharging @3
    SmartCharging @4

```

Fig. 4.7 Adding user's functions to *digexfun.def* export file

The next step consists in programming the new user-defined functions, which are called every time step from the DSL model. It is advisable to program them at the end of the source file *userfun.cpp*. There are two specific functions to exchange data between PowerFactory and user-defined functions: *pop()* and *push()*. On the one hand, the function *pop()* retrieves data from the DSL model. There should be as many *pop()* calling instances as input data are specified (variable *iargc*). Special care must be taken when data from *pop()* function calls are stored because it is sensitive to the call order. That is, data retrieval must be sorted in the same sequence as it has been defined when the user-defined function is called from the DSL block.

On the other hand, the *push()* function returns data to PowerFactory which will be loaded in the ongoing RMS simulation. Only one variable per time step can be returned. Thus, the *push()* function must be called once per user's function. Both functions only allow values of type double. Figure 4.8 shows the code that could be used for the uncontrolled charging of EVs.

After programming the new user's function, the *digexfun* project must be compiled and linked. It should be taken into account that any instance of PowerFactory program should be closed prior to compiling and linking the *digexfun* project. The reason is that PowerFactory blocks the access to the file *digexfun.dll* when it is active. Finally, the last step is calling the new user's function from a DSL block. If everything is correct, the name of the user's function must appear in blue after writing it in the correspondent DSL block, as shown in Fig. 4.9.

4.3.3 Interfacing with MATLAB API

From the *digexfun* project, it is possible to exchange data and execute MATLAB commands through the MATLAB API [3]. To achieve this aim, a previous configuration must be carried out in the Visual Studio project, mainly to indicate the libraries that are going to be used. As before, a 32 bits version of MATLAB must be considered. The steps are the following ones:

- (1) Add to the *project properties* → *configuration properties* → *VC++ directories* → *executable files* the following path: *C:\...\MATLABR...\bin\win32*

```

void __stdcall UncontrolledCharging(void){
    // Local variables declaration
    int id;
    double setpoint;

    // Pop input signals from DSL
    double t = pop();           // 1. Current simulation time
    double ev_id = pop();        // 2. EV ID
    double SOC = pop();          // 3. State of charge of the battery
    double aT = pop();           // 4. Arrival time of the EV
    double dT = pop();           // 5. Departure time of the EV
    double fSOC = pop();         // 6. Required final SOC (100% by default)
    double battCap = pop();      // 7. Battery capacity
    double cPower = pop();       // 8. Charging power rate
    double cEff = pop();         // 9. Charger efficiency

    // Conversion of type for EV identifier
    id = int(ev_id);

    // Initial conditions
    if(t<=0){
        iNEV = 0;
    }

    // Obtaining the total number of EVs in the network (iNEV is a global variable)
    if(id>iNEV){
        iNEV=id;
    }

    // No action outside the connection time of the EV
    if (t<aT || t>dT){
        setpoint = 0.0;
    }
    // Charge if the SOC of the battery is below the final required SOC
    else if (iNEV!=0){
        if(SOC<fSOC){
            setpoint = 0.0;
        }else{
            setpoint = 1.0;
        }
    }

    // Output of the function to DSL model
    push(setpoint);
}

```

Fig. 4.8 Example of a user's function for uncontrolled charging of EVs

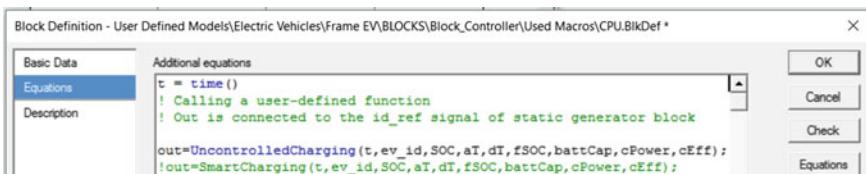


Fig. 4.9 User-defined function call from a DSL block

- (2) Add to the *project properties* → *configuration properties* → *VC++ directories* → *include files* the following path: *C:\...\MATLAB\R...\extern\include*
- (3) Add to the *project properties* → *configuration properties* → *VC++ directories* → *library files* the following path: *C:\...\MATLAB\R...\extern\lib\win32\microsoft*
- (4) Add to the *project properties* → *configuration properties* → *linker* → *input* → *additional dependencies* the following libraries: *libmx.lib*, *libeng.lib*, *libmex.lib* and *libmat.lib*
- (5) Finally, add to the environment variables of Windows the following path: *C:\...\MATLAB\R...\bin\win32*

After configuring the Visual Studio options, a set of code lines must be added to the header of *userfun.cpp*, as shown in Fig. 4.10.

Then, it is necessary to generate a pointer of MATLAB's engine inside the main function, as shown in Fig. 4.11. The command *engOpen("null")* will open a process of MATLAB on Windows system.

Once these steps have been carried out, MATLAB commands can be called from C++. The function *engEvalString* provides this functionality. A simple example is shown in the code of Fig. 4.12.

To have full functionality, data exchange between MATLAB and C++ program is required which can be carried out using two different functions: *engPutVariable* and *engGetVariable*. These two functions manage a specific data type called

```
#include <Engine.h>
#pragma comment (lib,"libmat.lib")
#pragma comment (lib,"libmx.lib" )
#pragma comment (lib,"libmex.lib")
#pragma comment (lib,"libeng.lib")
```

Fig. 4.10 Libraries to be loaded in the C++ program's header

```
Engine *m_pEngine
m_pEngine = engOpen("null");
// Checking the correct opening of Matlab
if (m_pEngine == "null"){
    printf("Error: Matlab linker",1);
}
```

Fig. 4.11 Opening an instance of MATLAB

```
//Creating a plot of a sine wave through digexfun interface
engEvalString(m_pEngine,"t=linspace(0,0.02,200);");
engEvalString(m_pEngine,"y=sin(2*pi*50*t);");
engEvalString(m_pEngine,"plot(t,y);");
engEvalString(m_pEngine,"grid on;");
```

Fig. 4.12 Invoking MATLAB commands from C++

```

double data_out[0];
mxArray* pf = mxCreateDoubleMatrix(1,1,mxREAL);
memcpy((void *)mxGetPr(pf),(void *)data_out,sizeof(double)*1);
engPutVariable(m_pEngine,"matlab_data",pf);
mxDestroyArray(pf);

```

Fig. 4.13 Passing a single value from C++ to MATLAB workspace

```

double data_out[SIZE];
mxArray* pf = mxCreateDoubleMatrix(1,SIZE,mxREAL);
memcpy((void *)mxGetPr(pf),(void *)data_out,sizeof(double)*SIZE);
engPutVariable(m_pEngine,"matlab_data",pf);
engEvalString(m_pEngine," matlab_data=matlab_data;");
mxDestroyArray(pf);

```

Fig. 4.14 Passing an array of values from C++ to MATLAB workspace

```

double data_out[SIZE_A][SIZE_B];
mxArray* pf = mxCreateDoubleMatrix(SIZE_A,SIZE_B,mxREAL);
memcpy((void *)mxGetPr(pf),(void *)data_out,sizeof(double)*SIZE_A*SIZE_B);
engPutVariable(m_pEngine,"matlab_data",pf);
engEvalString(m_pEngine," matlab_data=matlab_data;");
mxDestroyArray(pf);

```

Fig. 4.15 Passing a matrix of values from C++ to MATLAB workspace

mxArray, which is known and recognised by MATLAB and C++. Thus, an instance of this data type must be generated. The procedure is as follows:

- Create a *mxArray* type object and allocate the required memory space for it, using the command *mxCreateDoubleMatrix*.
- Copy the value to the instance of *mxArray* using the command *memcpy*.
- Provide the name of the variable to be generated in MATLAB workspace using the command *engPutVariable*.

Examples of passing data from C++ to MATLAB are shown in Fig. 4.13 for one single value, Fig. 4.14 for an array and Fig. 4.15 for a matrix.

An important advice when arrays or matrixes are passed from C++ to MATLAB is that the exchanged data must be transposed. This is because C++ has row-major

```

double result_var
double *cresult;
engEvalString(m_pEngine,"matlab_result= matlab_result'");
mxArray* mresult = engGetVariable(m_pEngine,"matlab_result");
cresult = mxGetPr(mresult);
result_var = *cresult
mxDestroyArray(mresult);

```

Fig. 4.16 Extracting results from MATLAB workspace to C++

storage while MATLAB has column-major storage. On the other hand, generated data structure *mxArray* should be destroyed using the function *mxDestroyArray* in order to free up memory space; otherwise, unexpected memory errors may happen during program execution. Finally, the inverse procedure is carried out when data from MATLAB is extracted. In this case, a pointer *cresult* is used to retrieve values from MATLAB API. It should be pointed out that a previous transpose of the data in MATLAB will be necessary. Figure 4.16 shows a simple example to retrieve a result of 1×1 dimension obtained in MATLAB API.

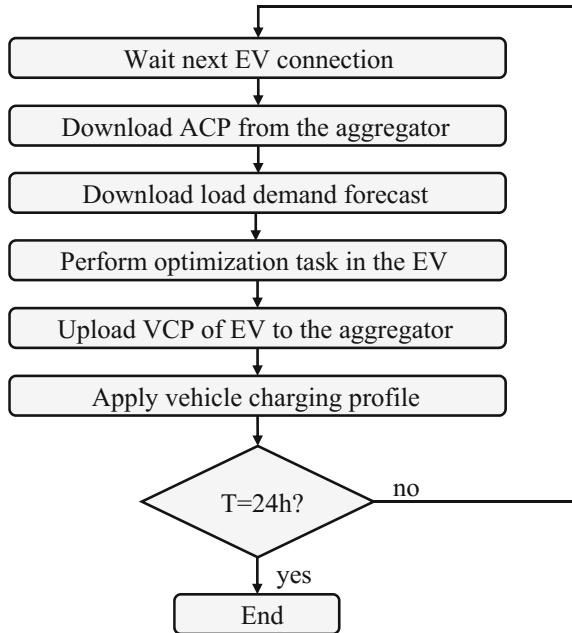
4.4 Example Project: Smart Charging of EVs

The following example demonstrates the usefulness of the digexfun interface to develop optimal-charging algorithms for EVs in low voltage distribution networks. The charge of a significant number of EVs could lead to several problems in distribution networks [4] such as increase of energy losses and load demand, overloads of lines and distribution transformers, voltage drops and unbalances and decrease of load factor. Researchers have proposed several possible solutions [5, 6], including methodologies based on multi-objective optimisation [7] and reactive power control [8], among others. Additionally, EVs can provide ancillary services such as primary frequency control [9].

One of the most common mistakes regarding optimal control-based solutions is that voltage levels are not analysed, mainly due to the complexity of simulating the charging of a large number of EVs, driven by an optimal control algorithm in a power system analysis software.

In this context, the following example shows the implementation of a smart charging algorithm method based on minimising the load variance of the distribution network. This way, load factor is increased, energy losses are limited and impacts on the low voltage distribution network are reduced. A test model of a residential low voltage distribution network is used. A 400 kVA distribution transformer supplies this radial network. Household load demand is defined following the reference [10] which has a resolution time of 5 min. The power factor of household loads is set to 0.95. 24-hour RMS simulations have been carried out (from 12:00 noon to 12:00 noon of the following day) for testing the adequacy of the presented smart charging strategy. The resolution of the optimal control algorithm is also 5 min. That is, charging set points of each EV are set every 5 min. Additionally, charging set point is considered continuous, so any charging power can be set as long as it is not higher than the rated power of the EV charger. Finally, EVs are modelled as constant power sources with a power factor of one.

Fig. 4.17 Flowchart of the optimal control algorithm



4.4.1 Optimal Control Algorithm

Implementing an optimal control algorithm based on minimising overall load variance requires the existence of an entity, usually named as aggregator or EV fleet manager, to coordinate the charging of EVs. The role of this aggregator in the present methodology is to aggregate the load demand of already connected EVs, that is, calculate the accumulated charging profile (ACP) of the EVs. When a new EV is connected to the network, the EV will receive this ACP and calculate the vehicle charging profile (VCP), to minimise the overall load variance of the distribution network. Figure 4.17 shows the simplified flowchart of the present example.

However, it is necessary to know in advance the energy demand profile of conventional loads. Thus, a daily load demand forecast must be done at distribution transformer level. In this example, a near perfect load forecast with a resolution time of one hour is assumed, as shown in Fig. 4.18. Nevertheless, load forecast errors must be taken into account in real implementations. In the near term, load forecast errors will be reduced due to massive use of smart metering devices.

The objective function to be minimised by each EV is presented in Eq. (4.1).

$$J = \frac{1}{dT} \sum_{t=aT}^{dT} [P_n \cdot x_{n,t} + ACP_t]^2 - \mu^2 \quad (4.1)$$

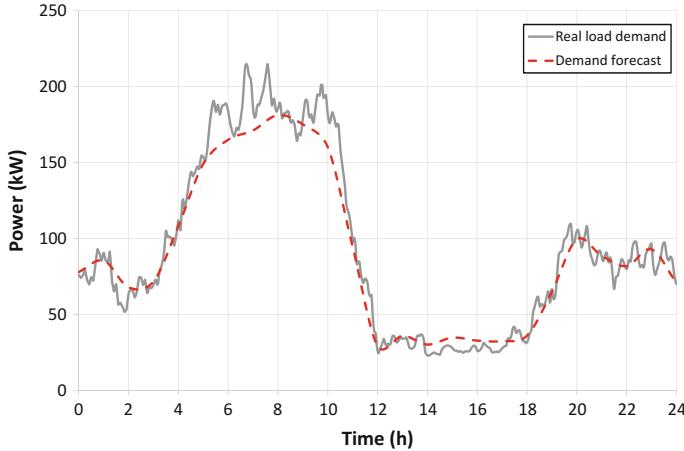


Fig. 4.18 Real load demand and load forecast used in the simulation

where n is the number of the EV (vehicle ID), t the time slot, aT and dT are the arrival and departure times, P_n is the rated charging power of the n -th EV, $x_{n,t}$ is the charging set point of the n -th EV during the time slot t , ACP_t is the accumulated charging profile of EVs at time slot t and μ is the average total demand of the considered period of time. The objective function of Eq. (4.1) is subject to several constraints to be met during the optimisation problem resolution. These constraints are the following ones:

- (1) SOC must be less than or equal to 1 (equivalent to 100% of SOC) at any time slot, as shown in Eq. (4.2).

$$\sum_{t=aT}^{dT} (P_n \cdot \eta_n \cdot x_{n,t}) \leq (Q_n - q_n) \cdot BC_n \quad \forall t \quad (4.2)$$

- (2) SOC must be equal to final SOC at final charging time dT , as shown in Eq. (4.3).

$$\sum_{t=aT}^{dT} (P_n \cdot \eta_n \cdot x_{n,t}) = (Q_n - q_n) \cdot BC_n \quad (4.3)$$

- (3) Variable $x_{n,t}$ must be between 0 and 1 (charging power limits), as shown in Eq. (4.4). This way, the obtained set point can be applied directly to DSL model in PowerFactory.

$$0 \leq x_{n,t} \leq 1 \quad \forall t \quad (4.4)$$

where η_n is the charging efficiency of the n -th EV, BC_n is the battery capacity of the n -th EV, Q_n is the final SOC required for the n -th EV and q_n is the initial SOC for the n -th EV.

As the objective function is quadratic, the quadratic programming technique is used, as shown in Eq. (4.5). The built-in function *quadprog* of the Optimisation Toolbox of MATLAB has been used to solve this optimisation problem.

$$\min_x \frac{1}{2} x' H x + f' x \quad \text{subject to} \quad \begin{cases} A_{ineq} \cdot x \leq b_{ineq} \\ A_{eq} \cdot x = b_{eq} \\ l_b \leq x \leq u_b \end{cases} \quad (4.5)$$

where x are the variables of the problem to be calculated, H is the symmetric matrix of quadratic coefficients of x , f is the vector of non-quadratic coefficients of x . On the other hand, A_{ineq} is the matrix of inequality constraints, b_{ineq} is the vector of inequality constraints, A_{eq} is the matrix of equality constraints, b_{eq} is the vector of equality constraints, and l_b and u_b are the lower and upper bounds of variable x .

In order to solve an optimisation problem such as the one presented here, a mathematical development should be carried out to obtain the required matrixes and vectors, which are passed to *quadprog* function, as shown in Eq. (4.6) where x is the final solution, $x0$ is the vector of initial solution of x and *opt* is an structure which defines the solver options.

$$x = \text{quadprog}(H, f, A_{ineq}, b_{ineq}, A_{eq}, b_{eq}, lb, ub, x0, \text{opt}) \quad (4.6)$$

4.4.2 Implementation Through Digexfun Interface

The implementation of an optimal control algorithm through digexfun interface is not trivial, due to the requirements in terms of space and organisation. However, the use of classes in C++ simplifies this task. Thus, it is always desirable to develop a class to manage the data of distributed resources such as EVs.

Figure 4.19 shows the declaration of the class used for this example. Note that *chargingProfile* variable is used to store the results obtained from the execution of the optimal control algorithm. The result has a resolution of 5 min, so 288 different charging set points are necessary to cover a 24-hour simulation.

Once the *class_ev* is defined, a global instance of it should be made in order to be usable from all functions programmed in *userfun.cpp*. Figure 4.20 shows the script for the smart charging control. The variance optimisation is carried out once per EV. However, a recursive calling of the optimal-charging control could be easily programmed. The EV charging set point for the current simulation time (or time slot) is obtained from the *chargingProfile* variable of the *class_ev* instance.

```

//----- Class EV header file (.hpp) -----
class class_ev {
public:
    // Constructor and destructor of the class
    class_ev();
    ~class_ev();
    //Parameters of the class EV
    bool optDone;           // Optimisation done flag
    int identifier;         // EV number
    double arrivalTime;    // Arrival time
    double departureTime;  // Departure time (set by user)
    double finalSOC;       // Final SOC required by user (100% by default)
    double battCapacity;   // Battery capacity
    double stateOfCharge; // SOC of the battery
    double chargePower;   // Nominal charging power
    double chargerEfficiency; // Charger efficiency
    double chargingProfile[288]; // Charging profile result (24hx60min/5min)
    // Declaration of class functions
    void SaveData (double id, double SOC,double battCap,double fSOC,double at, \
        double dT,double cPower,double cEff);
    void SetZero();
};

//----- Class EV source file (.cpp) -----
#include "class_ev.hpp"
// Constructor and destructor of the class
class_ev::class_ev(){
    ;
}
class_ev::~class_ev(){
    ;
}
// Function to save data of a specific EV
void class_ev::SaveData (double id,double fSOC,double battCap,double SOC, \
    double aT,double dT,double cPower,double cEff){
    identifier = int(id);
    finalSOC = fSOC;
    battCapacity = battCap;
    stateOfCharge = SOC;
    arrivalTime = int(aT);
    departureTime = int(dT);
    chargePower = cPower;
    chargerEfficiency = cEff;
}
// Function to initialize the instance of the class EV
Void class_ev::SetZero(){
    optDone = false;
    for (int i=0 ; i<288 ; i++){
        chargingProfile[i]=0.0;
    }
}

```

Fig. 4.19 Header and source code of the class which manages data of EVs

The optimisation problem is solved when the *varianceOptimisation* function is called.

The *varianceOptimisation* function (Fig. 4.21) is in charge of solving the optimisation problem through MATLAB API. Additionally, results of this optimisation are retrieved from MATLAB workspace, through the mxArray *mresult* and the *cresult* pointer. Each time a value of the charging profile has been retrieved, the pointer's address of variable *cresult* must be increased to obtain the following value.

```

void __stdcall SmartCharging(void){
    // Declaration of local variables
    double check;

    // Pop input signals from DSL
    t = pop();                                // 1. Current simulation time
    double ev_id = pop();                      // 2. EV ID
    double SOC = pop();                        // 3. State of charge of the battery
    double aT = pop();                         // 4. Arrival time of the EV
    double dT = pop();                         // 5. Departure time of the EV
    double fSOC = pop();                       // 6. Required final SOC
    double battCap = pop();                    // 7. Battery capacity
    double cPower = pop();                     // 8. Nominal charging power
    double cEff = pop();                       // 9. Charger efficiency

    // Conversion of type for EV identifier
    int id = int(ev_id);

    // Initial conditions of the simulation
    if(t<=0){
        init_cond(); // Function to initialize parameters and instances of class ev
    }
    // Obtaining the total number of EVs in the network
    if(id>1NEV){
        iNEV=id;
    }
    // Update the current time slot
    currentSlot = int(floor(t/(dTsim/timeSlots)));
    // Saving data of the current EV
    ev[id].SaveData(ev_id,SOC,battCap,fSOC,aT,dT,cPower,cEff);
    // No action outside the connection time of the EV
    if (t<aT || t>dT){
        ev[id].chargingProfile[currentSlot] = 0.0;
    }
    // If EV is connected...
    else if (iNEV!=0){
        // Checking if the optimal control algorithm has been executed or not
        if (ev[id].optDone==false){
            check = varianceOptimisation(id);           // Calling the optimal control function
            ev[id].optDone = true;                      // Mark as optimisation realized
        }
    }
    // Output of the function to the DSL model
    push(ev[id].chargingProfile[currentSlot]);
}

```

Fig. 4.20 Code for the smart charging function

The function *generate_data* is not detailed in this book chapter because it is mainly composed of multiple *for* loops; to generate all the required matrixes/vectors and the code to pass them to MATLAB workspace, a procedure has been already shown in Figs. 4.13, 4.14 and 4.15.

Finally, after coding, compiling and linking the digexfun project, the new user-defined function can be called from the DSL block, as shown in Fig. 4.22. In order to select between uncontrolled and smart charging functions, one of the call sentences must be commented, inserting an exclamation symbol in the block called CPU (located inside the electric vehicle DSL model). The output of this block must be connected to the input signal *id_ref* of the static generator block.

```

extern double varianceOptimisation(int id){
    // Declaration of local variables
    double charging_setpoint;
    double *cresult;
    double *check;

    // Create the different matrixes and vectors
    generate_data(id);

    // Setting the options of quadprog method in Matlab API
    engEvalString(m_pEngine,"opt=optimset('Algorithm','interior-point-convex');");
    // Executing quadprog function in Matlab API
    engEvalString(m_pEngine,"[x,v,f1]=quadprog(H,f,Aineq,bineq,Aeq,beq,lb,ub,[],opt);");

    // Check if there was any error during the optimisation problem resolution. In that
    // case, f1 parameter will be different to one
    mxArray* check_result = engGetVariable(m_pEngine,"f1");
    check = mxGetPr(check_result);
    if(*check!=1){
        // Return error number
        return(*check);
    }

    // Transposing results in Matlab API
    engEvalString(m_pEngine,"x=x';");

    // Retrieving the result of the optimisation from Matlab API
    mxArray* mresult = engGetVariable(m_pEngine,"x");
    cresult = mxGetPr(mresult);

    // Saving results in the charging profile of the EV
    for(int i=0 ; i<timeSlots ; i++){
        // Retrieve charging set point
        charging_setpoint=*cresult;
        // Save the charging profile of the EV
        ev[id].chargingProfile[i] = charging_setpoint;
        // Update ACP (in kW)
        ACP[i] += charging_setpoint*ev[id].chargePower;
        // Save the charging profile of the EV
        cresult++;
    }
    // Deleting mxArray instances
    mxDestroyArray(check_result);
    mxDestroyArray(mresult);

    // Output of the function
    return(1);
}

```

Fig. 4.21 Code for solving the variance minimisation problem

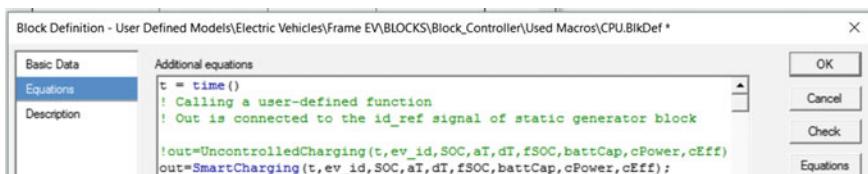
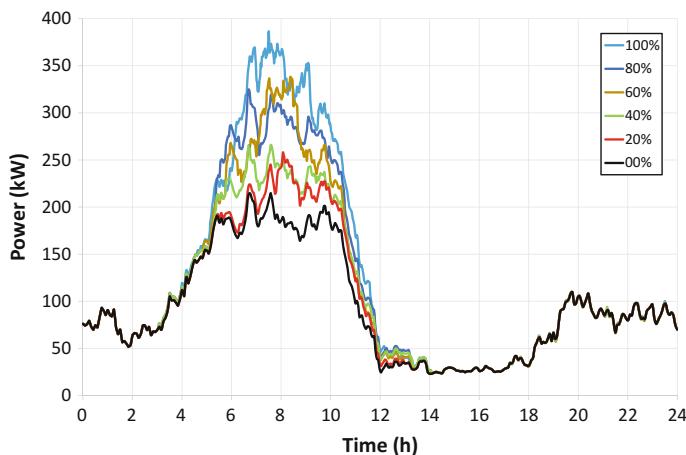


Fig. 4.22 Calling the *SmartCharging* function from the DSL block

Table 4.3 Data of different generated simulation scenarios

Penetration level (%)	Number of EVs	EVs per house	Model distribution (Leaf, Volt, Tesla)	EVs energy demand (kWh)
20	16	0.48	8,5,3	170
40	27	0.82	16,8,3	286
60	48	1.45	29,8,11	460
80	54	1.63	33,13,8	530
100	84	2.54	44,23,17	694

**Fig. 4.23** Distribution transformer load, for uncontrolled charging method

4.4.3 Simulation Results

Twenty-four-hour RMS simulations, for several EV penetration levels (Table 4.3), have been carried out in order to show the functionality of the developed optimised charging control method.

Figure 4.23 shows the load at transformer level for the different analysed cases using uncontrolled charging method. As it can be seen, uncontrolled charging of EVs has a great impact on the network because the charging of EVs is done at peak hours.

In contrast, the optimised charging control method reduces the load variance noticeably at transformer level, increasing the energy demand at off-peak hours (Fig. 4.24). As a result, the load factor of the system is improved. Therefore, energy losses are reduced compared to uncontrolled charging method.

Figure 4.25 and 4.26 showed the line–neutral voltage levels at the furthest node of the analysed network (node 838), for uncontrolled charging and optimised

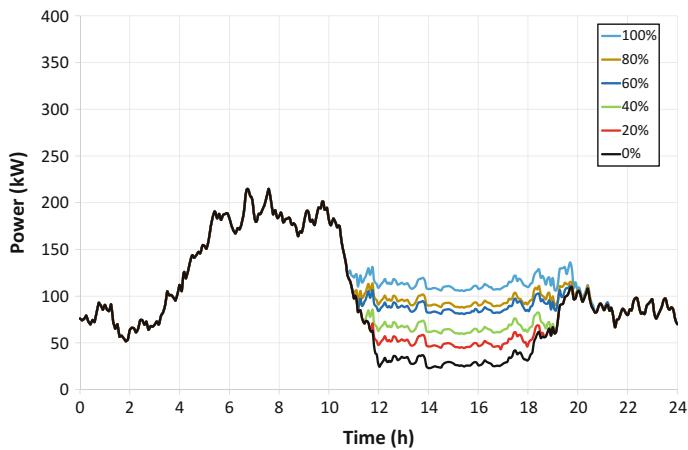


Fig. 4.24 Distribution transformer load, for the smart charging control

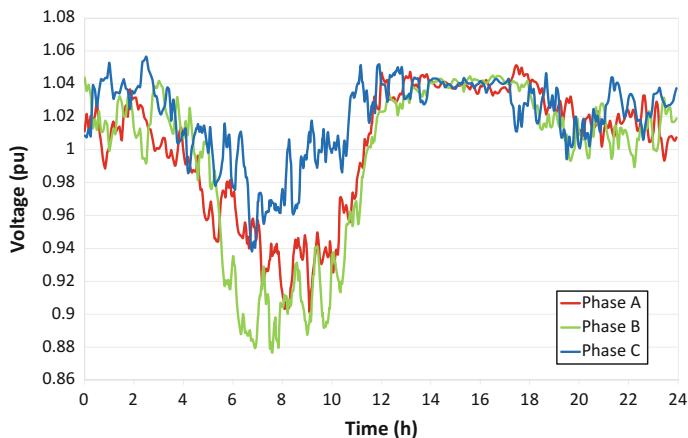


Fig. 4.25 Voltage levels, for uncontrolled charging method

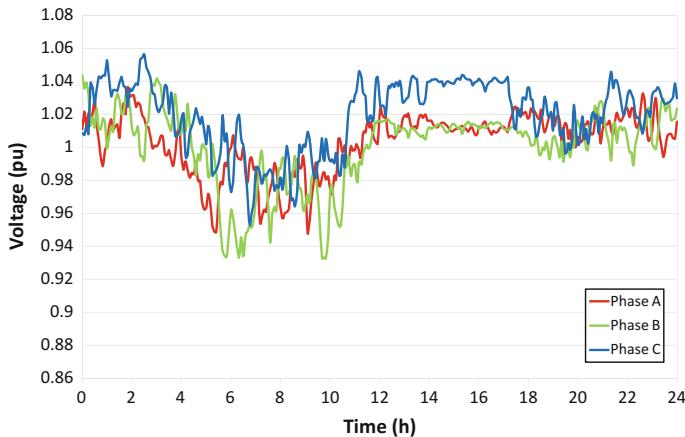


Fig. 4.26 Voltage levels, for optimised charging control

Table 4.4 Results obtained for the different cases

Case of study (EV penetration level) (%)	Load variance (kW^2)	Peak power (kW)	Energy losses (kWh)	Min. voltage (pu)
Base case (00)	2747	215	83	0.933
Uncontrolled (20)	3758	258	95	0.891
Variance opt. (20)	2350	215	85	0.933
Uncontrolled (40)	4518	264	100	0.910
Variance opt. (40)	2113	215	87	0.933
Uncontrolled (60)	8874	322	120	0.890
Variance opt. (60)	1878	215	92	0.933
Uncontrolled (80)	7164	336	120	0.892
Variance opt. (80)	1832	215	92	0.933
Uncontrolled (100)	11582	386	141	0.874
Variance opt. (100)	1794	215	98	0.933

charging control cases, respectively. The EV penetration level has been set to 100% in both cases. As can be seen in these figures, voltage levels are noticeably improved when the optimised charging control is used.

Finally, Table 4.4 summarises the numerical results of the analysed cases. Load variance has been calculated exporting the load demand data (HV side of the distribution transformer) to an Excel file. Load demand variance and energy losses are reduced in all cases when the proposed variance optimisation algorithm is applied. Specifically, a reduction of 31% in energy losses is achieved for the case of 100% EV penetration level.

4.5 Conclusions

Dealing with a large number of distributed energy resources, which are managed by optimal control methods, is not a trivial problem for power system analysis software tools. Fortunately, DIgSILENT PowerFactory software offers options to overcome this problem. On the one hand, DPL can be used to automatically generate and place elements in a distribution network. On the other hand, digexfun interface provides a way to develop a co-simulation framework between PowerFactory and MATLAB. The combination of both software tools is a powerful platform to test complex control algorithms for RMS/EMT simulations. In order to illustrate the operational readiness of this co-simulation framework, a smart charging algorithm for plug-in electric vehicles has been developed. The obtained results have proven the importance of the developed co-simulation framework to test optimal-charging control methods for plug-in electric vehicles.

Acknowledgements The work presented in this chapter has been supported by the Basque Government (group GISEL Ref. IT1083-16) and the University of the Basque Country—UPV/EHU (PES-12 and PPG 17/23).

References

1. DIgSILENT PowerFactory 15 user manual (2015)
2. J. Garcia-Villalobos, I. Zamora, P. Eguia, J.I. San Martin, F.J. Asensio, in *Modelling social patterns of plug-in electric vehicles drivers for dynamic simulations*, 2014 IEEE International Electric Vehicle Conference (IEVC), (2014), pp. 1–7
3. “MATLAB Engine API for C, C++, and Fortran.” [Online]. Available: <http://es.mathworks.com/help/matlab/calling-matlab-engine-from-c-c-and-fortran-programs.html>. Accessed: 05 June 2016
4. K. Clement-Nyns, E. Haesen, J. Driesen, The Impact of Charging Plug-In Hybrid Electric Vehicles on a Residential Distribution Grid. *IEEE Trans. Power. Syst.* **25**(1), 371–380 (2010)
5. J. García-Villalobos, I. Zamora, J.I. San Martín, F.J. Asensio, V. Aperribay, Plug-in electric vehicles in electric distribution networks: A review of smart charging approaches. *Renew. Sustain. Energy Rev.* **38**, 717–731 (2014)
6. J. Hu, H. Morais, T. Sousa, M. Lind, Electric vehicle fleet management in smart grids: a review of services, optimization and control aspects. *Renew. Sustain. Energy Rev.* **56**, 1207–1226 (2016)
7. J. García-Villalobos, I. Zamora, K. Knezović, M. Marinelli, Multi-objective optimization control of plug-in electric vehicles in low voltage distribution networks. *Appl. Energy* **180**, 155–168 (2016)
8. K. Knezović, M. Marinelli, Phase-wise enhanced voltage support from electric vehicles in a Danish low-voltage distribution grid. *Electr. Power Syst. Res.* **140**, 274–283 (2016)
9. M. Marinelli, S. Martinenas, K. Knezović, P.B. Andersen, Validating a centralized approach to primary frequency control with series-produced electric vehicles. *J. Energy Storage* **7**, 63–73 (2016)
10. I. Richardson, M. Thomson, D. Infield, C. Clifford, Domestic electricity use: A high-resolution energy demand model. *Energy Build.* **42**(10), 1878–1887 (2010)

Chapter 5

Probabilistic Load-Flow Using Analysis Using DPL Scripting Language



Francisco Gonzalez-Longatt, S. Alhejaj, A. Marano-Marcolini
and José Luis Rueda Torres

Abstract Load-flow analysis is an effective tool that is commonly used to capture the power system operational performance and its state at a certain point in time. Power grid operators use load-flow extensively on a daily basis to plan for day-ahead and dispatch scheduling among many other purposes. Also, it used to plan any grid expansion, alter or modernization. However, due to the deterministic nature and its applicability for only one set of operational data at a certain period, deterministic load-flow reduces the chances for predicting the uncertainty in power system. Researchers usually create a data model using probabilistic analyses techniques to produce a stochastic model that mimics the realistic system data. Combining this model with Monte Carlo methodology leads to form a probabilistic load-flow tool that is more powerful and potent to carry on many uncertainty tasks and other aspects of power system assessment. This chapter presents the

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_5) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

F. Gonzalez-Longatt (✉) · S. Alhejaj
School of Electronic, Electrical and Systems Engineering,
Loughborough University, Loughborough LE11 3TU, UK
e-mail: fglongatt@fglongatt.org

S. Alhejaj
e-mail: s.m.alhejaj@lboro.ac.uk

A. Marano-Marcolini
Department of Electrical Engineering, Universidad de Sevilla,
Camino de los Descubrimientos s/n, Seville, Spain
e-mail: alejandromm@us.es

J. L. Rueda Torres
Department of Electrical Sustainable Energy, Delft University
of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: J.L.RuedaTorres@tudelft.nl

DIgSILENT PowerFactory script language (DPL) implementation of a DPL script to perform probabilistic power flow (PLF) using Monte Carlo simulations (MCS) to consider the variability of the stochastic variables in the power system during the assessment of the steady-state performance. The developed PLF script takes input data from an external Microsoft Excel file, and then, the DPL can carry on a probabilistic load-flow and export the results using a Microsoft Excel file. The suitability of the implemented DPL is illustrated using the classical IEEE 14 buses.

Keywords DPL script · Simulation · Stochastic model · Probabilistic load-flow · Distribution function · Distribution density function · Monte Carlo

5.1 Introduction

The modern power systems are characterized by the introduction of more and more uncontrollable types of generation resources such as renewable generation (wind power, solar power, etc.); it increases the uncertainties which lead the more complex operation and control of the power system.

The increased level of uncertainties in the generation and demand side of the power system requires more efficient tools to be able to capture the variability in the system performance. A deterministic approach such as load-flow analysis is limited to a snapshot of steady-state power system operational data. Therefore, the *deterministic load-flow* (DLF) has an intrinsic limitation because it cannot have represented in a proper way the randomness in the power system.

Usually, a probabilistic approach is used when the deterministic approach is not able to capture the intrinsic variability. The *probabilistic approach* is one of the main three ways to capture and represent the intrinsic uncertainties in the power system beside *fuzzy arithmetic technique* and *interval mathematics*. The advantage and disadvantage of each of these approaches are discussed in [1].

The probabilistic approach of load-flow can be grouped into three approaches: Monte Carlo Simulation (MCS), analytical methods and approximation approaches. Monte Carlo simulation methodology is used widely and almost in all fields of engineering and science to simulate a repetitive process with different input data to produce different output results. In this chapter, a *DIgSILENT PowerFactory script language* (DPL) implementation of a DPL script to perform probabilistic power flow (PLF) using MCS is presented. This DPL is designed to consider the variability of the stochastic variables in the power system during the assessment of the steady-state performance. The PLF implemented in this chapter offers several advantages: (i) allow importing and exporting using structured database in the form of Microsoft Excel file (.xlsx), (ii) internal matrixes (*IntMatrix*) are used for communication between subscripts; it offers a fast communication and modular programming, allowing the re-use of the code into other applications.

The chapter starts with an introduction to probabilistic load-flow and its fundamentals, and then, an explanation about the Monte Carlo simulation and the main features of the DIgSILENT Simulation Language are presented. Then, the DPL implementation of the PLF is, and an illustrative example is presented.

5.2 Probabilistic Load-Flow: Fundamentals

The analysis of the steady-state conditions of a power system is one of the most commonly used tools in planning and operation of power systems. The classical load-flow is used to define the steady-state power balance in a power system: the total generation should be equal to the total power demand plus the power losses. It represents the so-called DLF; this tool is used to analyse and assess the planning and to operate the power system on a daily routine.

The DLF uses the known values of electrical power generation and power load demand of a selected network configuration to calculate the system states and power flows [2]. The formulation of the load-flow problem assumes that the data provided is absolutely precise and provides results totally compatible with the given data apart from round-off errors.

The integration of renewable generation units creates several planning and operation challenges. From the load-flow point of view, the random fluctuating character of wind speed causes the wind power plant (WPP) power production to be neither continuous nor slightly controllable.

The DPF analysis has the limitation of ignoring the grid uncertainties: outages, network changes, load variation. As a consequence, the deterministic approach is not sufficient for the analysis of modern power systems, the integration of renewable generation and other sources of randomness. The results of using DPF to attempt the calculation in power system considering uncertainties may lead to very different, even contradictory, or erroneous results, creating massive economic and technical consequences.

As expressed before, in the deterministic load-flow, the output of the model is fully determined by the parameter values and the initial conditions. An alternative approach is the use of stochastic models to represent the load-flow conditions considering uncertainties; this approach considers the inherent randomness; as a consequence, the same set of parameter values and initial conditions will lead to an ensemble of different outputs.

The typical *probabilistic load-flow* (PLF) analysis considers the power generation and grid configurations both to be discrete random variables, while load demand is a continuous random variable [3, 4]. Borkowska and Allan [5] proposed the stochastic load-flow (SLF) in 1974; it provides a full reflection of the influences of many factors' random variations in the power system. The PLF based on those methods directly treats the uncertainty of electric load, generation (especially wind power) and grid parameters. The term SLF is an alternatively used term for PLF and is generally favourable for system operational study that deals with short-term uncertainties [6].

The PLF methods can be included in three sets: (i) *analytical approach*, (ii) *numerical approach* and (iii) *approximate methods*.

Apart from the above grouping, *hybrid methods* uniting more than one of the above methods have attracted additional interest as they can overcome some of the limitations of the individual constituting methods.

The analytical approach analyses a system and its inputs using mathematical expressions, i.e. using convolution techniques, with *probabilistic density functions* (PDF) of stochastic variables of power inputs so that PDFs of stochastic variables of system states and line flows can be obtained. Details of this approach can be found on [5–8]. Analytical methods include techniques such as *convolution method*, *cumulant method* (CM).

The numerical approach and sampling methods include techniques such as MCS, Latin hypercube sampling, uniform design sampling.

5.3 Monte Carlo Simulation Applied to the Load-Flow Problem

In general, the DLF problem consists in finding the zero of a set of nonlinear equations starting defining the power system power balance from an adequate initial guess.

The most general form of the load-flow equations is a set of *differential-algebraic equations* (DAE) in steady state [9]. Then, the formulation of the power flow equations is reduced to the algebraic:

$$\begin{aligned} \mathbf{g}(\mathbf{x}) &= \mathbf{0} \\ [\mathbf{g}^P(\mathbf{x}) &\quad \mathbf{g}^Q(\mathbf{x})]^T = \mathbf{0} \end{aligned} \tag{5.1}$$

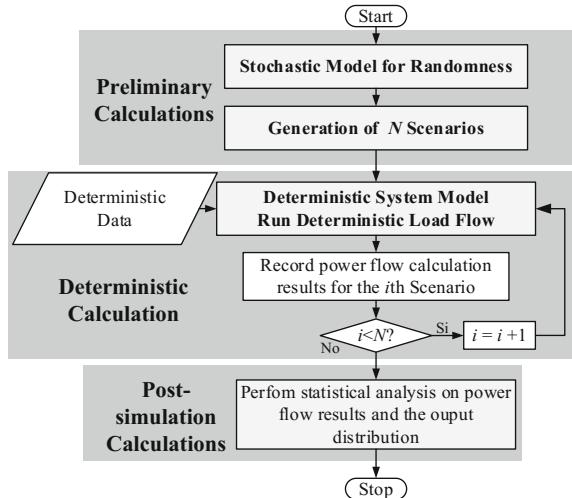
where \mathbf{g} is the set of algebraic equations that define the power balance at network buses and \mathbf{x} the state vector. For classical formulation of AC DPF, the *inputs* or known quantities are the injected active powers (P_i) at all busbars (where P and Q or P and V are known) except the slack bus; the injected reactive powers (Q_i) at all load busbars (where P and Q are known); and the voltage magnitude at all generator busbars (where P and V are *outputs* or known).

$$\begin{aligned} P_i &= g_i^P(\delta_1, \delta_2, \dots, \delta_n, V_1, V_2, \dots, V_n) \\ Q_i &= g_i^Q(\delta_1, \delta_2, \dots, \delta_n, V_1, V_2, \dots, V_n) \end{aligned} \tag{5.2}$$

where $i = 1, 2, \dots, n$. n represents the number of power buses, nonlinear voltage (V) and phase (δ) relationships. A complete explanation for the classical AC power flow can be found on [10–12].

PLF attempts to obtain PDFs of state vector and line flows of a statistically varying electrical network [13]. P_i and Q_i , are considered by their distributions, usual with binomial repartition with p_i and q_i the probability of up, respectively down state for each unit generation $P_{g,i}$ and the PDF load $P_{L,i}$ is continuous and normal with m (mean) and σ (standard deviation) [14–16]. MCS is a method for iteratively evaluating a deterministic model using sets of random numbers as inputs [17]. Stochastic power flow (SPF) is solved using MSC, which involves repeating

Fig. 5.1 Flowchart of MCS applied to solve PPF



the DLF simulation process using in each simulation a particular set of values of the random variables (loads, conventional and wind generation productions at each node of the considered power system).

MSC is used to solve PLF; it involves repeating the DPF calculation process using in each simulation a particular set of values of the random variables, called *simulation scenario*. Depending upon the number of uncertainties and the ranges specified for them, a MSC could involve a large number of scenarios and recalculations before it is complete. Figure 5.1 shows details of the process of solving Probabilistic power flow (PPF) using MTC.

This chapter adopts a *Monte Carlo* (MC) method for the SPF analysis. This technique is used in [18] to solve the SPF problem including wind farms by repeated simulations. The two main features of MCS are as follows: (a) it provides considerably accurate results, but the computation time is consuming for large systems with several uncertain parameters, (b) it can be easily combined with pre-existent DPF programs to create and easy and fast implementation of SPF. In this paper, the approach selected is an SPF based on MSC.

5.4 DIgSILENT Programming Language (DPL)

Before starting to delve into the program and coding,¹ it is worth explaining how *DIgSILENT Programming Language* (DPL) works in the context of the modern programming world. DPL is a scripting language that enables the users to do many

¹In order to use DPL, users are required to have some programming experience and writing code in some of the modern used programming languages such as C++, Java and/or Python with some good understanding of the main concepts of object-oriented programming (OOP).

automated tasks available in DIgSILENT PowerFactory [19]. Therefore, the users will have full control over the power grid parameters, power system calculations and pre/post processing the information coming from power system studies. Additionally, the DPL is built with the object-orientated concept in mind so that everything that the user can deal with can be considered as a set of classes that have some parameters and functionalities too. The main and principal class is the DPL command class² [1].

The probabilistic power flow implemented in this chapter is implemented using the previously mentioned MSC approach. The DPL implementation assumes the simulation scenarios consist of a number of samples generated per each random variable represented in the problem. For simplicity, the implementation assumed the scenarios are provided to the DPL using a database with a well-defined data structure, taking the opportunity of the DPL capability of managing Microsoft Office files, the script imports the scenarios from a Microsoft Excel file and then the results are exported using the same type of files.

5.5 DPL Implementation: Probabilistic Load-Flow

The implementation of the PLF using MSC consists of one main script which is able to call four subscripts. DPL main script is named ‘‘**ProbabilisticLF**’’ and contains the main logic behind the probabilistic load-flow; it involves: (i) importing the simulation scenarios to be used in the MCS, (ii) calculating the deterministic load-flow per each scenario and (iii) export the results of the deterministic load-flow.

The input of the DPL implementation consists of a database containing ($N_{samples}$) simulation scenarios including the variables related to the system stochasticity. The implementation of this chapter considers the variability on the active (P) and reactive power (Q) on loads, generators, wind power plants, PV plants and PHEV.

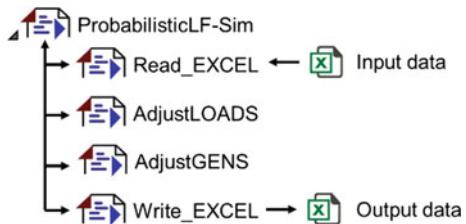
The implementation of the PLF is created using a modular approach where main activities are directed from the main script, and the subroutines perform very specific actions. A matrix approach is used for communication and data interchange between the subroutines and main script. Matrices (*IntMat*) are considered as an external object of the main script (**StochData.xlsx**), but also the subroutines are considered internal objects; this approach allows the main script and subroutines use the same matrices to exchange data and information.

The main DPL, ‘‘**ProbabilisticLF**’’, includes four subscripts (see Fig. 5.2):

- ‘‘**Read_MSExcel**’’ is designed to open the MS Excel file and read the data contained in the file considering the specific data structure (for more details see

²This component is well explained and documented in the DPL manual of the DIgSILENT PowerFactory software.

Fig. 5.2 Logical structure of the ProbabilisticLF.ComDpl



the file named “**StochData.xlsx**”). When this subscript is executed, twelve matrices (*IntMat*) are loaded with the scenarios data: active power (**DATA_P_ESS**, **DATA_PG**, **DATA_PL**, **DATA_P_PHEV**, **DATA_P_PV**, **DATA_P_WPP**) and reactive power (**DATA_Q_ESS**, **DATA_QG**, **DATA_QL**, **DATA_Q_PHEV**, **DATA_Q_PV**, **DATA_Q_WPP**).

- “**Read_Matrix**”: this subroutine is used to read the input matrix (matrixA) and copies it into the output matrix (matrixB).
- “**Adjust_Loads**”: the subroutine is designed to adjust the values of active (p_{lini}) and reactive power (q_{lini}) at the load elements (ElmLod). The script made use of a general selection (Set) to access the specific load where the variability is considered.
- “**Adjust_Gens**”: this subroutine is designed to adjust the values of the active (p_{gini}) and reactive power (q_{gini}) of all the power sources and storage devices in the power network. The subroutine uses a general selection where all the power sources and energy storage equipment are stored: synchronous machines, wind turbines, electric vehicles, PV systems and energy storage. The power network uses *ElmSym* to model the classical synchronous generators and *ElmGenstat*, static generator element for modelling the power converter-based technologies.
- “**Write_Excel**”: this subroutine is designed to write the data results of the probabilistic load-flow into a Microsoft Excel file with a very specific data structure.
- The next subsection of this chapter is dedicated to explaining in basic and generic terms the programming aspects of each of the scripts above.

5.6 Main Script: ProbabilisticLF

The script named “**ProbabilisticLF**” represents the main program of the PLF implementation, and it is programmed in a very modular way where the subscripts can be systematically called and using matrix (*IntMatrix*) to interchange data and allow the flow of information inside the program.

For simplicity, the main variables involved in the probabilistic load-flow are defined as input parameters of the “**ProbabilisticLF**” DPL command. Figure 5.2

shows the main input parameters of the probabilistic simulation including the filename and path of the input and output data, a number of elements to be considered as statistically defined (elements where the active and reactive power will change during the Monte Carlo simulation). Also, the authors have included few configurations related to the load-flow analysis function (*ComLdf*). A flag (*iopt_net*) is used to allow the probabilistic load-flow analysis considered the power flow: (0) balanced, (1) unbalanced or (2) using the DC load-flow method.

The main script **ProbabilisticLF.ComDPL** uses external objects as a way to create an artificial communication between the main script and subscripts (see Fig. 5.3). Figure 5.4 shows the name, object and description of the twelve matrix objects (*IntMat*) used by the probabilistic load-flow implementation main script and its subscripts.

A key aspect of the modular programming approach used in this implementation is the fact that all the objects are inside the main script: (i) subscripts in the form of DPL commands and (ii) matrices (*IntMat*). Because all the objects and subscripts are located inside, the same folder allows a horizontal communication with the matrixes allowing a simple data interchange. All the objects inside the main command DPL, “**ProbabilisticLF**”, can be observed suing the button *Contents* of the main command DPL dialogue box. Figure 5.5 shows the contents of the **ProbabilisticLF** command DPL, where the subscripts and matrix objects are shown.

ProbabilisticLF-Sim					
Study Cases\Base Case\DPL Commands Set					
Input parameters:					
	Type	Name	Value	Unit	Description
1	string	OUTPUT_FILENAME	E:\MYPCLMy Desktop\Sto		Output Data Filename MS Excel
2	string	INPUT_FILENAME	E:\MYPCLMy Desktop\Sto		Input Data Filename MS Excel
3	int	Nsample	10000		Number of Samples
4	int	PF_Mode	0		Operation mode: 0 Balanced, 1: Unbalanced
5	int	NWPP	1		Number of Wind Power Plants
6	int	NESS	1		Number of Energy Storage Systems
7	int	NPV	2		Number of PV Systems
8	int	NPHEV	1		Number of PHEV
9	int	NGEN	5		Number of Synch Generators
10	int	NLOAD	11		Number of Loads

Fig. 5.3 ProbabilisticLF.ComDpl: lists of the input parameters

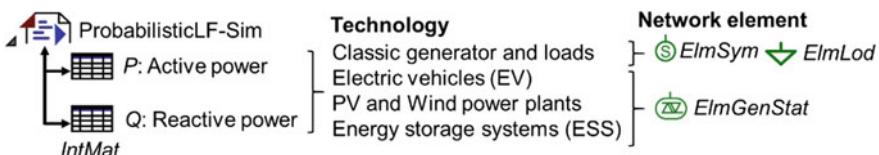


Fig. 5.4 Interaction between ProbabilisticLF.ComDpl and the external object and its interaction with network objects

External Objects:			
	Name	object	Description
► 1	PESS	PESS	Active power of the Energy Storage System
2	QPV	QPV	Reactive power of the PV systems
3	PPV	PPV	Active power of the PV systems
4	QWPP	QWPP	Reactive power of the Wind Power Plants
5	QPHEV	QPHEV	Reactive power of the PHEV
6	QESS	QESS	Rective power of the Energy Storage Systems
7	PWPP	PWPP	Active power of Wind Power Plants
8	PG	PG	Active power of Synchronous Machines
9	PL	PL	Active power of loads
10	PPHEV	PPHEV	Active power of the PHEV

Fig. 5.5 ProbabilisticLF.ComDpl: list of the external objects

In the sequel to this section, a discussion of all tasks is carried through this main script that is illustrated with the relative segment of code.

The DPL script language uses a syntax quite similar to the C++ programming language. This type of language is intuitive, easy to read and easy to learn. The starting point of the main script is the variable definition section; it allows to declare the local variables and initialize them (if needed); see Fig. 5.6.

The next segment of code is used to quantify the type of network elements and the number of them in the active project (see the snippet shown in Fig. 5.7). The script capture for a particular network element, all the object and store them in memory. For example, *AllRelevant()* method is called to get all the load elements '**.ElmLod*' in the grid and save them in a set of a variable called '*S_load*'. The next programming line, "*n_load = S_load.Count();*" determine the number of objects in the set. Similarly, similar lines of code are written to get the set of all instants of all other network elements and their number count in the active network.

The system information is shown on the output window using the snippet shown in Fig. 5.8.

The main script starts calling the first subscript *READ_EXCEL* in order to import the data from the Microsoft Excel file and save them to into the RAM memory using matrixes. DPL is an object-oriented program, and it requires a precise definition of the variables. The imported data is stored inside ten matrices; five are used for active power and five for reactive power; the size of each matrix is internally initialized using the object method *Init* (see Fig. 5.9); it initializes the matrix with given size and values, regardless of the previous size and data.

The database is imported from the Microsoft Excel file; the file name and path are store in the string named *INPUT_FILENAME_EXCEL*, and it is declared as input parameter of the main script. Then, the filename of the input date is assigned to the *FILENAME* parameter and transferred to the *READ_EXCEL* subscript. Also, the number of samples *Nsample* is also passed to that subscript (see Fig. 5.10). At that point, the subscript *READ_EXCEL* is executed to import all the data records in the excel worksheets to the internal matrices: **DATA_P_ESS**, **DATA_PG**, **DATA_PL**, **DATA_P_PHEV**, **DATA_P_PV**, **DATA_P_WPP**) and reactive

- Scripts\ProbabilisticLF-Sim :

	Name	Order	Type	Object modified	Object modified
▶	AdjustLOADS	-1000000		02/05/2017 20:41:44	FranciscoM
▶	AdjustGENS	-1000000		02/05/2017 20:41:44	FranciscoM
▶	ReadMATRIX	-1000000		02/05/2017 20:41:44	FranciscoM
▶	Read_EXCEL	-1000000		02/05/2017 20:41:44	FranciscoM
▶	Write_EXCEL	-1000000		08/05/2017 17:50:34	FranciscoM
█	DATA_PG			02/05/2017 20:41:44	FranciscoM
█	DATA_PL			02/05/2017 20:41:44	FranciscoM
█	DATA_P_ESS			02/05/2017 20:41:44	FranciscoM
█	DATA_P_PHEV			02/05/2017 20:41:44	FranciscoM
█	DATA_P_PV			02/05/2017 20:41:44	FranciscoM
█	DATA_P_WPP			02/05/2017 20:41:44	FranciscoM
█	DATA_QG			02/05/2017 20:41:44	FranciscoM
█	DATA_QL			02/05/2017 20:41:44	FranciscoM
█	DATA_Q_ESS			02/05/2017 20:41:44	FranciscoM
█	DATA_Q_PHEV			02/05/2017 20:41:44	FranciscoM
█	DATA_Q_PV			02/05/2017 20:41:44	FranciscoM
█	DATA_Q_WPP			02/05/2017 20:41:44	FranciscoM
█	lol			02/05/2017 20:41:44	FranciscoM
█	LOAD			02/05/2017 20:41:44	FranciscoM
█	LOADING			02/05/2017 20:41:44	FranciscoM
█	LinePac			02/05/2017 20:41:44	FranciscoM
█	LinePdc			02/05/2017 20:41:44	FranciscoM
█	Plosses			02/05/2017 20:41:44	FranciscoM
█	Qess			02/05/2017 20:41:44	FranciscoM
█	Qgen			02/05/2017 20:41:44	FranciscoM
█	Qgenstat			02/05/2017 20:41:44	FranciscoM
█	Qphev			02/05/2017 20:41:44	FranciscoM
█	Qpv			02/05/2017 20:41:44	FranciscoM
█	Qvsc			02/05/2017 20:41:44	FranciscoM
█	Qwind			02/05/2017 20:41:44	FranciscoM
█	VBUS			02/05/2017 20:41:44	FranciscoM
█	per			02/05/2017 20:41:44	FranciscoM

Ln 1 32 object(s) of 32 1 object(s) selected

Fig. 5.6 List the objects contained in the main command DPL: ProbabilisticLF.ComDpl

```
! Probabilistic Load Flow (PLF) using Monte Carlo Simulations
! Created by: Dr Francisco Gonzalez-Longatt, April 2015
! Variable Definitions
! Variable Definitions
int      error, iter, ii, n_bus, n_line, n_load;
int      n_sym, n_genstat, n_vscmono, n_wind, n_PV, n_ESS, n_PHEV;
double   M, LO, u1;
double   Time_o, Time_f, Dpl_time;
object   PF, O1, O2, O3, O4, O5, O6, O7, O8, O9, O10;
set     Bus, SB, S_bus, S_line, S_load, S_Sym, S_Genstat, S_VSCmono;
set     sWind, sPV, sESS, sPHEV;
```

Fig. 5.7 Snippet ProbabilisticLF.ComDpl: variable definitions

```

! ----- Get Network Elements-----
! Loads: ElmLod
S_load = AllRelevant('*.ElmLod'); ! Returns a set of all available loads in the grid
o1 = S_load.First(); ! Returns the first object in the set
n_load = S_load.Count(); ! Returns the number of loads in the grid
! Buses: ElmTerm
S_bus = AllRelevant('*.ElmTerm'); ! Returns a set of (ElmTerm) objects (buses)
o2 = S_bus.First(); ! Returns the first object of set.
n_bus = S_bus.Count(); ! Returns the number of buses
! Lines: ElmLne
S_line = AllRelevant('*.ElmLne'); ! Returns a set of all (ElmLne) objects (lines)
o3 = S_line.Firstmatch('ElmLne'); ! Returns the first object in the set
n_line = S_line.Count(); ! Returns the number of lines
! Synchronous Generators: ElmSym
S_Sym = AllRelevant('*.ElmSym'); ! Returns a set of (ElmSym) objects (Synch Mach.)
o4 = S_Sym.Firstmatch('ElmSym'); ! Returns the first object in the set
n_sym = S_Sym.Count(); ! Returns the number of Synch. Mach. objects
! Wind Power plants: ElmGenstat
sWind = AllRelevant('WF*.ElmGenstat'); ! Returns a set of all (ElmGenstat) objects
o7=sWind.Firstmatch('WF*.ElmGenstat'); ! Returns the first object in the set
n_wind = sWind.Count(); ! Returns the number of objects (wind farms)
! PV Power plants: : ElmGenstat
sPV = AllRelevant('*ElmPvsy'); ! Returns a set of all (ElmPvsy) objects (PV)
o8 = sPV.Firstmatch('ElmPvsy'); ! Returns the first objects of ElmPvsy.
n_pv = sPV.Count(); ! Returns the number of PV objects
! ESS -Energy Storage Systems: ElmGenstat
sESS = AllRelevant('Storage.ElmGenstat'); ! Returns a set of (ElmGenstat) objects
o9 = sESS.First(); ! Returns the first objects Storage.ElmGenstat.
o9.ShowFullName();
n_ESS = sESS.Count(); ! Returns the first objects Storage.ElmGenstat.
! PHEV -Electric Vehicle: : ElmGenstat
sPHEV = AllRelevant('PHEV.ElmGenstat'); ! Returns a set (PHEV.ElmGenstat) objects
o10=sPHEV.Firstmatch('PHEV.ElmGenstat'); ! Returns the first object in this set
n_PHEV = sPHEV.Count();

```

Fig. 5.8 Snippet ProbabilisticLF.ComDpl: get the network elements

```

printf(' PROBABILISTIC LOAD FLOW \n');
printf(' RUNNING MONTE-CARLO SIMULATIONS ');
printf(' created by Prof. F Gonzalez-Longatt and Samir Alhejaj, Sep 2016');
printf(' ----- ');
printf(' Number of Buses : %2.2f',n_bus);
printf(' Number of Loads : %2.2f',n_load);
printf(' Number of Lines : %2.2f',n_line);
printf(' Number of Synchronous Generators : %2.2f',n_sym);
printf(' Number of Wind Turbines : %2.2f',n_wind);
printf(' Number of PV Plant : %2.2f',n_pv);
printf(' Number of ESS : %2.2f',n_ESS);
printf(' Number of PHEV : %2.2f',n_PHEV);

```

Fig. 5.9 Snippet ProbabilisticLF.ComDpl: show the power system and network elements information

power (**DATA_Q_ESS**, **DATA_QG**, **DATA_QL**, **DATA_Q_PHEV**, **DATA_Q_PV**, **DATA_Q_WPP**). The internal structure and operation of the subscript **READ_EXCEL** are explained in next subsections.

The main script communicates between subscripts using matrices; as a consequence, the reader must recognize that the imported data from Microsoft Excel is

```

DATA_PL.Init(Nsample,NLOAD); ! Initialize an internal matrix for Load
DATA_QL.Init(Nsample,NLOAD);
DATA_PG.Init(Nsample,NGEN); ! Initialize an internal matrix for Syn Gen
DATA_QG.Init(Nsample,NGEN);
DATA_P_WPP.Init(Nsample,NWPP); ! Initialize an internal matrix for WPP
DATA_Q_WPP.Init(Nsample,NWPP);
DATA_P_PV.Init(Nsample,NPV); ! Initialize an internal matrix for PVP
DATA_Q_PV.Init(Nsample,NPV);
DATA_P_ESS.Init(Nsample,NESS); ! Initialize an internal matrix for ESS
DATA_Q_ESS.Init(Nsample,NESS);
DATA_P_PHEV.Init(Nsample,NPHEV); ! Initialize an internal matrix for PHEV
DATA_Q_PHEV.Init(Nsample,NPHEV);

```

Fig. 5.10 Snippet ProbabilisticLF.ComDpl: initialize the values of the internal matrixes

transferred into specific matrices for the internal use inside the main script. Figure 5.11 shows the command lines to copy the data in the “PG” matrix into the “matrixA” matrix of “ReadMATRIX” subscript. By calling “ReadMATRIX” subscript and execute it, the data are returned and saved into “DATA_PG” matrix. Similarly, the same is executed for all other matrices. The internal structure and operation of the subscript “ReadMATRIX” are explained in next subsections.

Now, the main script initiates the Monte Carlo simulation process. Initially, set the start time of the simulation processing time (see Fig. 5.12). It also initializes the “PF” object for load-flow class “ComLdf” by calling “GetCaseObject()”. This object contents all the details that are necessary for running the load-flow analysis.

The Monte Carlo simulation process is a systematic process where the deterministic load-flow is evaluated by each of the scenarios defined in the *Nsamples*. As a consequence, a loop is used to counter “iter” the number of the scenarios; the loop “for” will complete the calculation of “Nsampel” defined in the input parameter of the “ProbabilisticLF-Sim” DPL command. The simulation process is simple; inside the loop, the value of the active and reactive power of the elements considered as stochastically described is updated, and then, a deterministic load-flow is calculated (see Fig. 5.13).

The subscripts “AdjustLOADS” and “AdjustGENS” are used to set the active and reactive power of the passive and active network elements. The internal structure and operation of the subscript “AdjustLOADS” and “AdjustGENS” are explained in next subsections.

A health check is included in the Monte Carlo simulation process. The script executes the load-flow analysis using the following sentence “error = PF.Execute ()”; if the load-flow is successful and there is coverage, the execution function will return 0 and if not will return 1. The user can add error handling routine to display an error message in case there is no convergence and load-flow does not complete.

```

! Importing Data From Microsoft Excel File (INPUT_FILENAME_EXCEL.xls)
READ_EXCEL:FILENAME = INPUT_FILENAME_EXCEL;
READ_EXCEL:Nsamples = Itermax;
READ_EXCEL.Execute();

```

Fig. 5.11 Snippet ProbabilisticLF.ComDpl: importing data from Microsoft Excel

```

! ----- Populate the Internal Matrices -----
ReadMATRIX:matrixA = PG;           ! Copy Synch. Gen. active power data into matrixA
ReadMATRIX:matrixB = DATA_PG;      ! Assign matrix DATA_PG to matrixB
ReadMATRIX.Execute();
ReadMATRIX:matrixA = QG;           ! Swap data from PG into DATA_PG
ReadMATRIX:matrixB = DATA_QG;      ! Copy Synch. Gen. reactive power data to matrixA
ReadMATRIX.Execute();
ReadMATRIX:matrixA = PL;           ! Assign matrix DATA_QG to matrixB
ReadMATRIX:matrixB = DATA_PL;      ! Swap data from QG into DATA_QG
ReadMATRIX.Execute();
ReadMATRIX:matrixA = QL;           ! Copy Load active power data to matrixA
ReadMATRIX:matrixB = DATA_PL;      ! Assign matrix DATA_PL to matrixB
ReadMATRIX.Execute();
ReadMATRIX:matrixA = QL;           ! Swap data from PL into DATA_PL
ReadMATRIX:matrixB = DATA_QL;      ! Copy Load reactive power data to matrixA
ReadMATRIX.Execute();
ReadMATRIX:matrixA = PWPP;          ! Assign matrix DATA_QL to matrixB
ReadMATRIX:matrixB = DATA_P_WPP;    ! Swap data from QL into DATA_QL
ReadMATRIX.Execute();
ReadMATRIX:matrixA = QWPP;          ! Copy WPP active power data to matrixA
ReadMATRIX:matrixB = DATA_Q_WPP;    ! Assign matrix DATA_P_WPP to matrixB
ReadMATRIX.Execute();
ReadMATRIX:matrixA = QWPP;          ! Swap data from PWPP into DATA_P_WPP
ReadMATRIX:matrixB = DATA_Q_WPP;    ! Copy QWPP reactive power data to matrixA
ReadMATRIX:matrixB = DATA_Q_WPP;    ! Assign matrix DATA_Q_WPP to matrixB
ReadMATRIX.Execute();
ReadMATRIX:matrixA = PPV;           ! Swap data from QWPP into DATA_Q_WPP
ReadMATRIX:matrixB = DATA_P_PPV;    ! Copy PPV active power data to matrixA
ReadMATRIX.Execute();
ReadMATRIX:matrixA = PPV;           ! Assign matrix DATA_P_PPV to matrixB
ReadMATRIX:matrixB = DATA_Q_PPV;    ! Swap data from PPV into DATA_P_PPV
ReadMATRIX:matrixA = QPV;           ! Copy QPV reactive power data to matrixA
ReadMATRIX:matrixB = DATA_Q_PPV;    ! Assign matrix DATA_Q_PPV to matrixB
ReadMATRIX.Execute();
ReadMATRIX:matrixA = QPV;           ! Swap data from PG into DATA_Q_PPV
ReadMATRIX:matrixB = DATA_P_ESS;    ! Copy PESS active power data to matrixA
ReadMATRIX:matrixB = DATA_Q_ESS;    ! Assign matrix DATA_P_ESS to matrixB
ReadMATRIX.Execute();
ReadMATRIX:matrixA = QESS;          ! Swap data from PESS into DATA_P_ESS
ReadMATRIX:matrixB = DATA_Q_ESS;    ! Copy QESS reactive power data to matrixA
ReadMATRIX:matrixB = DATA_Q_ESS;    ! Assign matrix DATA_Q_ESS to matrixB
ReadMATRIX.Execute();
ReadMATRIX:matrixA = PPHEV;          ! Swap data from QESS into DATA_Q_ESS
ReadMATRIX:matrixB = DATA_P_PHEV;   ! Copy PHEV active power data to matrixA
ReadMATRIX:matrixB = DATA_P_PHEV;   ! Assign matrix DATA_P_PHEV to matrixB
ReadMATRIX.Execute();
ReadMATRIX:matrixA = QPHEV;          ! Swap data from PG into DATA_PHEV
ReadMATRIX:matrixB = DATA_Q_PHEV;   ! Copy QPHEV reactive power data to matrixA
ReadMATRIX:matrixB = DATA_Q_PHEV;   ! Assign matrix DATA_Q_PHEV to matrixB
ReadMATRIX.Execute();

```

Fig. 5.12 Snippet ProbabilisticLF.ComDpl: populate internal matrices for the Monte Carlo simulation process

```

! MONTE CARLO SIMULATION PROCESS
! Initializing variables for Monte-Carlo simulations
Time_o = GetTime(4);           ! Get System Time
PF = GetCaseObject('.ComLdf'); ! Returns first found object of '*.ComLdf' class
                                ! from the currently active study case
PF:iopt_net = PF_Mode;         ! PF_Mode = 1 force unbalanced
                                ! = 0 force balanced

```

Fig. 5.13 Snippet ProbabilisticLF.ComDpl: adjustment and setting of the deterministic load-flow inside the Monte Carlo simulation process

In the end of the load-flow analysis, which will take few seconds, new results values will be saved to each element result object by PowerFactory (see Fig. 5.14).

The deterministic load-flow results are obtained per each simulation scenario; internal matrixes are used to collect the relevant results: bus voltages, lines loading, current in each transmission line and power losses of the transmission lines (internal matrices: *VBUS*, *LOADING*, *Iol* and *Plosses*; see Fig. 5.14).

```

117. ! -----Start Monte Carlo Simulation -----
118. for (iter = 1; iter <= Itermax; iter+= 1) {
119. ! BEGIN iter
120. printf(' -----');
121. printf(' Scenario #%',iter);
122. ! 2.a. ADJUSTING THE STOCHASTIC ELEMENTS
123. ! Adjust Loads ElmLod
124. AdjustLOADS: iID = iter;
125. AdjustLOADS: oMatrixP = DATA_PL;
126. AdjustLOADS: oMatrixQ = DATA_QL;
127. AdjustLOADS.Execute();
128. ! Adjust Synch Generators ElmSym
129. AdjustGENS: iID = iter;
130. AdjustGENS: sGenType= S_Sym;
131. AdjustGENS: oMatrixP = DATA_PG;
132. AdjustGENS: oMatrixQ = DATA_QG;
133. AdjustGENS.Execute();
134. ! Adjust Wind Power plants ElmGenstat
135. AdjustGENS: iID = iter;
136. AdjustGENS: sGenType= SWind;
137. AdjustGENS: oMatrixP = DATA_P_WPP;
138. AdjustGENS: oMatrixQ = DATA_Q_WPP;
139. AdjustGENS.Execute();
140. ! Adjust PV power plants ElmGenstat
141. AdjustGENS: iID = iter;
142. AdjustGENS: sGenType= SPV;
143. AdjustGENS: oMatrixP = DATA_P_PV;
144. AdjustGENS: oMatrixQ = DATA_Q_PV;
145. AdjustGENS.Execute();
146. ! Adjust ESS ElmGenstat
147. AdjustGENS: iID = iter;
148. AdjustGENS: sGenType= sESS;
149. AdjustGENS: oMatrixP = DATA_P_ESS;
150. AdjustGENS: oMatrixQ = DATA_Q_ESS;
151. AdjustGENS.Execute();
152. ! Adjust PHEV ElmGenstat
153. AdjustGENS: iID = iter;
154. AdjustGENS: sGenType= sPHEV;
155. AdjustGENS: oMatrixP = DATA_P_PHEV;
156. AdjustGENS: oMatrixQ = DATA_Q_PHEV;
157. AdjustGENS.Execute();

```

Fig. 5.14 Snippet ProbabilisticLF.ComDpl: adjustment in the active and reactive power of the active and passive network elements

The numerical results of the load-flow calculations are exported into Microsoft Excel file; the output data is transferred by using the “**Write_EXCEL**” subscript. The results exporting process start by saving the “Loading” percentage of the transmission lines data to the first worksheet and giving it a name “Loading”. Similarly, the buses voltages “Voltages”, lines currents “Currents”, lines active power “Pij” and power losses “Plosses” are saved (see Fig. 5.15). After the systematic calling of the “**Write_EXCEL**” subscript, the whole set of data is exported into a single Microsoft Excel file and the Monte Carlo simulation process ends.

```

! EXECUTE DETERMINIST LOAD FLOW (ComLdf)
error = PF.Execute();           ! Executes the command Load Flow
! COLLECT ALL RELEVANT DATA
! I. Bus Voltages (per unit) -ElmTerm
02 = S_bus.First();            ! Returns the first matching object ElmTerm
for(ii = 1; ii <= n_bus; ii+= 1) {
M = 02:m:u1;                  ! u1: Magnitude of Terminal Voltages
VBus.Set(iter,ii,M);          ! Set the value at position (i,ii) in the matrix
02 = S_bus.Next();             ! Returns the next matching object Elmterm
}
! II. Line Loading Conditions (%) -ElmTerm
03 = S_line.First();           ! Returns the first matching object ElmTerm
for(ii=1; ii <= n_line; ii+=1) {
LO = 03:c:loading;            ! loading: %
LOADING.Set(iter,ii,LO);
03 = S_line.Next(); }
! III. Synchronous Generator -ElmSym
04 = S_Sym.First();
for(ii=1;ii<=n_sym;ii+=1) {
LO = 04:m:Q:bus1;             ! Q: reactive power generation
Qgen.Set(iter,ii,LO);
04 = S_Sym.Next(); }
! V. Wind Power Plant -ElmGenstat
07 = sWind.First();            ! Returns the first objects ElmGenstat
for(ii=1;ii<=n_wind;ii+=1) {
LO = 07:m:Q:bus1;              ! Q: reactive power generation
Qwind.Set(iter,ii,LO);
07 = sWind.Next(); }
! VI. PV Power Plant - ElmGenstat
08 = sPV.First();              ! Returns the first objects ElmGenstat
for(ii=1;ii<=n_PV;ii+=1) {
LO = 08:m:Q:bus1;              ! Q: reactive power generation
Qwind.Set(iter,ii,LO);
08 = sPV.Next(); }
! VII. ESS -ElmGenstat
09 = sESS.First();             ! Returns the first objects ElmGenstat
09.ShowFullName();
for(ii=1;ii<=n_ESS;ii+=1) {
LO = 09:m:u:bus1;              ! Q: reactive power generation
Qess.Set(iter,ii,LO);
09 = sESS.Next(); }
! VIII. PHEV -ElmGenstat
010 = sPHEV.First();            ! Returns the first objects ElmVSCmono
010.ShowFullName();
for(ii=1;ii<=n_PHEV;ii+=1) {
LO = 010:m:u:bus1;
Qess.Set(iter,ii,LO);
010 = sESS.Next(); }
! IX. Line Current Iij -ElmLne
03 = S_line.First();            ! Returns the first matching object ElmTerm
for(ii=1;ii<=n_line;ii+=1) {
LO = 03:m:i:bus1;
Iol.Set(iter,ii,LO);           ! Save the Loading value to LOADING Matrix
03 = S_line.Next(); }
! XI. Lines AC Power flow Pij -ElmLne
03 = S_line.First();            ! Returns the first matching object ElmTerm
for(ii=1;ii<=n_line;ii+=1) {
M = 03:m:P:bus1;
LinePac.Set(iter,ii,M);         ! Save the Loading value to PinBus Matrix
03 = S_line.Next(); }
! XII. Active power losses -ElmLne
03 = S_line.First();            ! Returns the first matching object ElmTerm
for(ii=1;ii<=n_line;ii+=1) {
M = 03:m:Ploss:bus1;
Plosses.Set(iter,ii,M);         ! Save the Loading value to PinBus Matrix
03 = S_line.Next(); }
} ! END Iter
!

```

Fig. 5.15 Snippet ProbabilisticLF.ComDpl: executing deterministic load-flow and collecting the numerical results

5.7 DPL Subroutines

The main script **ProbabilisticLF.ComDpl** includes four subscripts (see Fig. 5.1):

- “**Read_Excel**”: is designed to open the MS Excel file and read the data contained in the file considering the specific data structure (for more details, see the file named “**StochData.xlsx**”).
- “**Read_Matrix**”: this subroutine is used to read the input matrix (*matrixA*) and copies it into the output matrix (*matrixB*).
- “**Adjust_Loads**”: the subroutine is designed to adjust the values of active (*plini*) and reactive power (*qlini*) at the load elements (*ElmLod*). The script made use of a general selection (*Set*) to access the specific load where the variability is considered.
- “**Adjust_Gens**”: this subroutine is designed to adjust the values of the active (*pgini*) and reactive power (*qgini*) of all the power sources and storage devices in the power network.
- “**Write_Excel**”: this subroutine is designed to write the data results of the probabilistic load flow into a Microsoft Excel file with a very specific data structure.

The next subsection of this chapter is dedicated to explain in basic and generic terms the programming aspects of each of the aforementioned scripts.

5.7.1 *Reading Stochastic Data from Excel File: Read_Excel*

DIgSILENT PowerFactory allows the communication with Microsoft Office; as a consequence, the basic features of all spreadsheets in Microsoft Excel can be used to read and write data. The subscript named **Read_EXCEL** is used to load the simulation scenarios (*Nsamples*) used in the probabilistic load-flow; the data records of active and reactive powers for active and passive network components are stored in a structure MS Excel file. Then, the subroutine DPL script **Read_EXCEL** reads the data from the file and stores the read data into internal matrices (*IntMatrix*).

The subscript uses as input parameter a string variable (*FILENAME*) receiving the name and location of the MS Excel file that holds the stochastic data, and an integer variable is also declared to be assigned the number of scenarios to be simulated (*Nsamples*)—see Fig. 5.16. Moreover, numbers of global matrices are defined to hold the data. Such variables can be added by clicking on the “Contents” button from the “Basic Options” tab of the DPL command dialogue (Fig. 5.17).

When the subscript is called, it prints the file name “*stochData.xlsx*” and its path. In the case of any problem with the excel software or the excel data file, an error message will be displayed as part of the error handling functionality available in DPL. The number of worksheets available in the excel file is read by “*xlGetWorksheetCount()*” method and assigned to an integer variable “*iCount*”.

```

! EXPORTING SIMULATION RESULTS TO MICROSOFT EXCEL SHEETS
! 1. Saving Loading Results
Write_EXCEL: oResult = LOADING;
Write_EXCEL: sResult = 'Loading';
Write_EXCEL:sheetIndex = 1;
Write_EXCEL.Execute();
! 2. Saving Bus Voltages Results
Write_EXCEL: oResult = VBUS ;
Write_EXCEL: sResult = 'Voltages';
Write_EXCEL:sheetIndex = 2;
Write_EXCEL.Execute();
! 3. Saving Line current Results
Write_EXCEL: oResult = Iol ;
Write_EXCEL: sResult = 'Currents';
Write_EXCEL:sheetIndex = 3;
Write_EXCEL.Execute();
! 4. Saving Lines Active Power (AC) Results
Write_EXCEL: oResult = LinePac ;
Write_EXCEL: sResult = 'Pij';
Write_EXCEL:sheetIndex = 4;
Write_EXCEL.Execute();
! 5. Saving Power Losses Results
Write_EXCEL: oResult = Plosses ;
Write_EXCEL: sResult = 'Power losses';
Write_EXCEL:sheetIndex = 5;
Write_EXCEL.Execute();
! END OF EXPORTING SIMULATION RESULTS TO MICROSOFT EXCEL SHEETS
Time_f=GetTime(4);           ! GetSystemTime();
Dpl_time = Time_f - Time_o;
printf('Processor Time: %g sec',Dpl_time);

```

Fig. 5.16 Snippet ProbabilisticLF.ComDpl: executing deterministic load-flow and collecting the numerical results

The screenshot shows the 'Read_EXCEL' dialog box. At the top, there is a 'Name' field containing 'Read_EXCEL'. Below it is a 'General Selection' section with three buttons: a downward arrow, an upward arrow, and an ellipsis (...). Underneath is a 'Input parameters:' section. A table lists two parameters:

	Type	Name	Value	Unit	Description	
► 1	string	FILENAME			Filename MS Excel	↑
2	int	Nsamples			Number of Samples	

Fig. 5.17 Details of the input parameters of the Read_EXCEL.ComDpl

A loop goes through each worksheet and reads the data from and saves them to the right matrix. For example, the first worksheet data will be read and assigned to “PG” matrix by “Set()” method and so on for the other matrices. Reading data from excel worksheet is well documented and illustrated by some examples from the knowledge base of the DIgSILENT software website. After the end of importing data process, “xITerminate()” method closes the current instant of excel file (see Figs. 5.18 and 5.19).

- Scripts\ProbabilisticLF-Sim\Read_EXCEL :

	Name	Order	Type	Object modified	Object modified
▶	PESS			02/05/2017 20:41:44	FranciscoM
▶	PG			02/05/2017 20:41:44	FranciscoM
▶	PL			02/05/2017 20:41:44	FranciscoM
▶	PPHEV			02/05/2017 20:41:44	FranciscoM
▶	PPV			02/05/2017 20:41:44	FranciscoM
▶	PWPP			02/05/2017 20:41:44	FranciscoM
▶	QESS			02/05/2017 20:41:44	FranciscoM
▶	QG			02/05/2017 20:41:44	FranciscoM
▶	QL			02/05/2017 20:41:44	FranciscoM
▶	QPHEV			02/05/2017 20:41:44	FranciscoM
▶	QPV			02/05/2017 20:41:44	FranciscoM
▶	QWPP			02/05/2017 20:41:44	FranciscoM

Fig. 5.18 Internal matrices used to store the data read by the subscript Read_EXCEL

5.7.2 Coping Matrices Content Between Two Subscripts: **ReadMATRIX**

The subscript **Read_MATRIX** is designed to copy the data records that are been saved to **READ_EXCEL** subscript matrices into the main program script matrices. The **ReadMATRIX** subscript transforms the local data read from the excel file into a global dataset, and then, it can be used by other subscripts. Even this task seems an additional and not required; the subscript is used here for illustration purposes. Users can still use a different approach or using the passing arguments and getting results through methods that are provided by DPL. The main task here is to swap data from matrix “*matrixA*” to matrix “*matrixB*” and then return the last matrix into the main script (Fig. 5.20).

5.7.3 Adjusting Loads: **Adjust_LOADS**

The subscript named **Adjust_LOADS** is called from the main script in order to set the simulation scenario; it is done by adjusting the active and reactive power in each passive element of the network for the iterations of Monte Carlo simulation process. The network elements are selected by using “*AllRelevant()*” method. A counter “*jj*” is set to the first object “*oObj*” of these elements, and a loop is iterating through each object of this element and assign active power “*plini*” and reactive power “*qlini*” (see Fig. 5.21).

```

! DPL Subscript - Read Data From Excel File
! VARIABLE DEFINITION
int iError, iCount, i, iRow, iCol, iStop, iLstr;
double dValue;
string sString, sString1;
printf(' Loading MS Excel Filename : %s ',FILENAME);
iError = xlStart(); ! Creates a new MS Excel instance
if(iError) {
    Error(' Unable to start MS Excel application '); exit(); }
! Opens an existing workbook = opens xls file
iError = xlOpenWorkbook(FILENAME);
if (iError) {
    Error(' Unable to open Excel file ');
    xlTerminate(); ! Closes currently active MS Excel instance
    exit(); }
iCount = xlGetWorksheetCount(); ! Get number of sheets
printf(' Number of Sheets : %i',iCount);
for(i = 1; i<= iCount; i=i+1) {
    xlActivateWorksheet(i); sString = xlGetWorksheetName(i);
    iRow = 1; iStop = 0;
    while(iStop = 0) {
        iCol = 1;
        while(1) {
            xlGetValue(iCol, iRow, sString1); xlGetValue(iCol, iRow, dValue);
            iLstr = strlen(sString1); ! Returns the length of a string
            if (iLstr = 0) { ! Stop at empty cell, continue with next row
                if (iCol = 1) {
                    iStop = 1; ! Completely stop if cell in first column is empty
                }
                break;
            }
            if (i=1) { PG.Set(iRow,iCol,dValue); }
            if (i=2) { QG.Set(iRow,iCol,dValue); }
            if (i=3) { PL.Set(iRow,iCol,dValue); }
            if (i=4) { QL.Set(iRow,iCol,dValue); }
        ! Adding the power generation sources
            if (i=5) { PWPP.Set(iRow,iCol,dValue); }
            if (i=6) { QWPP.Set(iRow,iCol,dValue); }
            if (i=7) { PPV.Set(iRow,iCol,dValue); }
            if (i=8) { QPV.Set(iRow,iCol,dValue); }
            if (i=9) { PESS.Set(iRow,iCol,dValue); }
            if (i=10) { QEES.Set(iRow,iCol,dValue); }
            if (i=11) { PPHEV.Set(iRow,iCol,dValue); }
            if (i=12) { QPHEV.Set(iRow,iCol,dValue); }
            iCol = iCol + 1; }
        iRow = iRow + 1; }
}
xlTerminate(); ! Closes currently active MS Excel instance

```

Fig. 5.19 Snippet Read_EXCEL.ComDpl

5.7.4 Adjusting Generators Power: *Adjust_GENS*

The subscript named **Adjust_GENS** is called from the main script in order to set the simulation scenario; it is done by adjusting the active and reactive power in each active element of the network for the iterations of Monte Carlo simulation process (see Fig. 5.22).

```

! DPL Subscript - Swapping The Contents For Two Matrices
int nrowsA, ncolsA, nrowsB, ncolsB, i1, j1;
double VALUE;
printf(' LOADING MATRIXES');
nrowsA = matrixA.NRow();    ncolsA = matrixA.NCol();
nrowsB = matrixB.NRow();    ncolsB = matrixB.NCol();
VALUE = 1;
for(i1 = 1; i1 <= nrowsA; i1 = i1 + 1) {
    for(j1 = 1; j1 <= ncolsA; j1 = j1 + 1) {
        VALUE = matrixA.Get(i1,j1); matrixB.Set(i1,j1,VALUE);
    }
}

```

Fig. 5.20 Snippet Read_MATRIX.ComDpl

```

! DPL Subscript - Adjusting Loads
int iN, jj, ii;
double a, b;
string sLname;
object oObj;
set sLoads;
printf(' ADJUSTING LOADS'); printf(' Adjusting Iteration : %i',iID);
sLoads = AllRelevant('*ElmLoad');
iN = sLoads.Count();
printf(' Number of Elements in General Selection : %i',iN);
printf(' Full Name of the Objects');
oObj = sLoads.First();
jj = 0;
! Cycle through the objects in the set and print out the full name
while(oObj) {
    jj = jj+1;
    a = oMatrixP.Get(iID,jj);
    oObj:plini = a;
    b = oMatrixQ.Get(iID,jj);
    oObj:qlini = b;
    sLname = oObj:loc_name;
    printf(' %i. %s: Pl = %3.3f MW Ql = %3.3f MVAR ', jj, sLname,a,b);
    oObj = sLoads.Next();
}

```

Fig. 5.21 Snippet Adjust_LOADS.ComDpl

5.7.5 Exporting Results Data into Excel File: Write_EXCEL

The subscript named Write_EXCEL is called from the main script in order to export the numerical results of the Monte Carlo simulation process into a single Microsoft Excel file. This subscript creates a new excel file and gives it uses as a name the input parameter *OUTPUT_FILENAME_EXCEL* entered in the main script. Then, it iterates through each row and column value (using “*mxValue()*” method) in the matrix and writes it (using “*xlSetValue()*” method) to the correspondence cell in the excel worksheet (see details in Fig. 5.23).

```

! DPL Subscript - Adjusting Generation Values
int iN, jj, ii;
double a,b;
string sGname;
object oObj;
printf(' ADJUSTING GENS'); printf(' Row to be adjusted : %i', iID);
iN = sGenType.Count(); printf(' Number of Elements in General Selection : %i',iN);
printf(' Full Name of the Objects');
oObj = sGenType.First();
jj = 0;
! Cycle through the objects in the set and print out the full name
while(oObj) {
    jj = jj+1;
16.     a = oMatrixP.Get(iID,jj);
17.     oObj:pgini = a ;
18.     b = oMatrixQ.Get(iID,jj);
19.     oObj:qgini = b ;
20.     sGname = oObj:loc_name;
21.     printf(' %i. %s: Pg = %3.3f MW Qg = %3.3f MVAR', jj, sGname, a, b);
22.     oObj = sGenType.Next();
23. }

```

Fig. 5.22 Snippet Adjust_GENS.ComDpl

5.8 Simulations and Results

The classical IEEE 14 bus test system is used in this chapter to illustrate the results of the probabilistic load-flow. The IEEE 14 bus test system represents a portion of the American electric power system (in the Midwestern US) as of February, 1962. The test system consists of five synchronous machines; three of which are synchronous compensators used only for reactive power support. There are 11 loads in the system totalling 259 MW and 81.3 Mvar. The data of the IEEE 14 bus test system is publically available at: https://www2.ee.washington.edu/research/pstca/pf14/pg_tca14bus.htm.

The IEEE 14 bus test system is used in this section, and it has been modelled using DIgSILENT PowerFactory. However, the model used here is customized variation of the original system which is created to integrate new technologies where the presence of uncertainties could be modelled using probabilistic data. New generation technologies, energy storage system and electric vehicles are included. Two wind power plants are added: onshore wind farm (WF1) and offshore wind farm (WF2). A multi-terminal HVDC system (three terminals) is used to connect the WF1 and WF2 into the IEEE 14 bus system. Two photovoltaic plants (PV1 and PV2), electric vehicle (PHEV) and battery energy storage system BESS (battery energy storage system) are added to the network. Table 5.1 lists all the generation elements in the grid with their nominal ratings. Figure 5.23 shows the single line diagram of the customized IEEE-14 test model, and Fig. 5.24 shows details of the HVDC transmission network use to integrate the wind farm power plants. The customized IEEE 14 bus test system is a representative network of the future power system considering several new technologies.

```

! Write2EXCEL: Subroutine designed to write the results of the probabilistic load flow
! into a Microsoft Excel file.
! VARIABLE DEFINITION
int iError, savingErr, noRows, noCols, row, col, noOfXleWSs, xlWSno;
double mxValue;
string sWSName, sNAME;
int results, ii, n_bus, n_line;
object O2, O3;
set S_bus, S_line;
printf(' Data Transfer to Excel File Started ...');
xlAddWorksheet(sResult); ! Create New Worksheet
! ADDING HEADERS
results = strcmp(sResult,'Voltages');
if (results ==0) {
    S_bus = AllRelevant('.ElmTerm'); ! Returns a set with calculation relevant objects
    O2 = S_bus.First();           ! Returns the first objects ElmTerm.
    n_bus = S_bus.Count();        ! Returns the number of stored ElmTerm.
    for(ii = 1; ii <= n_bus; ii+= 1) {
        sNAME = O2:loc_name;
        O2 = S_bus.Next();       ! Returns the next matching object Elmterm
        xlSetValue(ii,1,sNAME);
    }
}
if (results <>0) {
    S_line = AllRelevant('.ElmLne'); ! Returns a set with calculation relevant objects
    O3 = S_line.Firstmatch('ElmLne'); ! Returns the first objects ElmLne.
    n_line = S_line.Count();
    for(ii = 1; ii <= n_line; ii+= 1) {
        sNAME = O3:loc_name;
        O3 = S_line.Next();       ! Returns the next matching object Elmterm
        xlSetValue(ii,1,sNAME);
    }
}
printf(' Excel-Worksheet name: %s', sResult );
! Reading Data from Matrix and write it to Excel
noRows = oResult.NRow(); noCols = oResult.NCol();
mxValue = 0;
for (row = 1 ; row <= noRows ; row =row+1) {
    for (col = 1 ; col <= noCols ; col= col+1) {
        mxValue = oResult.Get(row,col);
        xlSetValue(col,row+1,mxValue); }
}

```

Fig. 5.23 Snippet Write_EXCEL.ComDpl

Seven load elements are connected to the test system, and Table 5.2 shows the active and reactive power demand of the loads (Fig. 5.25).

5.9 Stochastic Data Modelling

The IEEE 14 bus test system has been customized to include new technologies of generation, energy storage and also transportation (electric vehicles). Those new technologies and the loads are used to include uncertainties in the system in order to demonstrate the use of the probabilistic load-flow. The uncertainties are modelled in this paper using probabilistic models described by a probability distribution.

Table 5.1 Details of the installed capacity of the active elements in the test system

Generator	Type	Bus	Installed capacity (MVA)
G1	Synchronous generator	1	304.00
G2	Synchronous generator	2	304.00
G3	Synchronous generator	3	80.00
WF2	Offshore wind farm	4, 5	6.15
CS6	Synchronous capacitor	6	55.00
CS8	Synchronous capacitor	8	55.00
PHEV	Power hybrid electric vehicle	9	1.00
Storage	Energy storage (battery)	10	5.00
PV2	Photovoltaic plant	11	5.00
PV1	Photovoltaic plant	12	2.50
WF1	Onshore wind farm	14	6.52

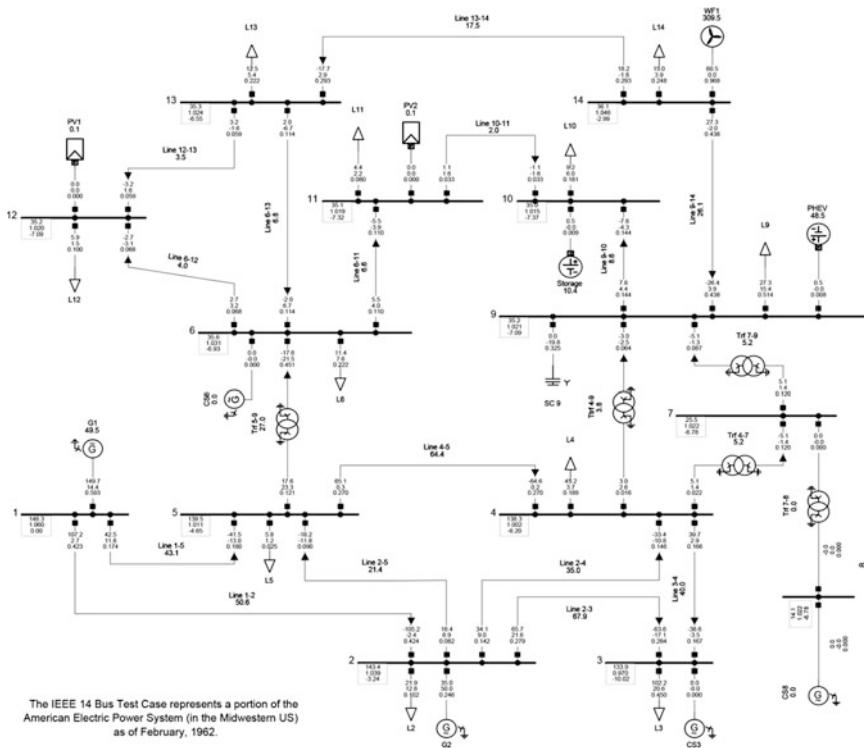
**Fig. 5.24** Customized IEEE 14 bus test system including new technologies

Table 5.2 Details of the power consumption on the load connected in the test system

Load	Bus	Active power (MW)	Reactive power (MVar)
L2	2	21.95	12.85
L3	3	102.22	20.62
L4	4	45.14	3.69
L5	5	5.80	1.22
L6	6	11.39	7.63
L9	9	27.34	15.38
L10	11	4.35	2.24
L11	11	4.35	2.24
L12	12	5.88	1.54
L13	13	12.46	5.35
L14	14	14.99	3.86

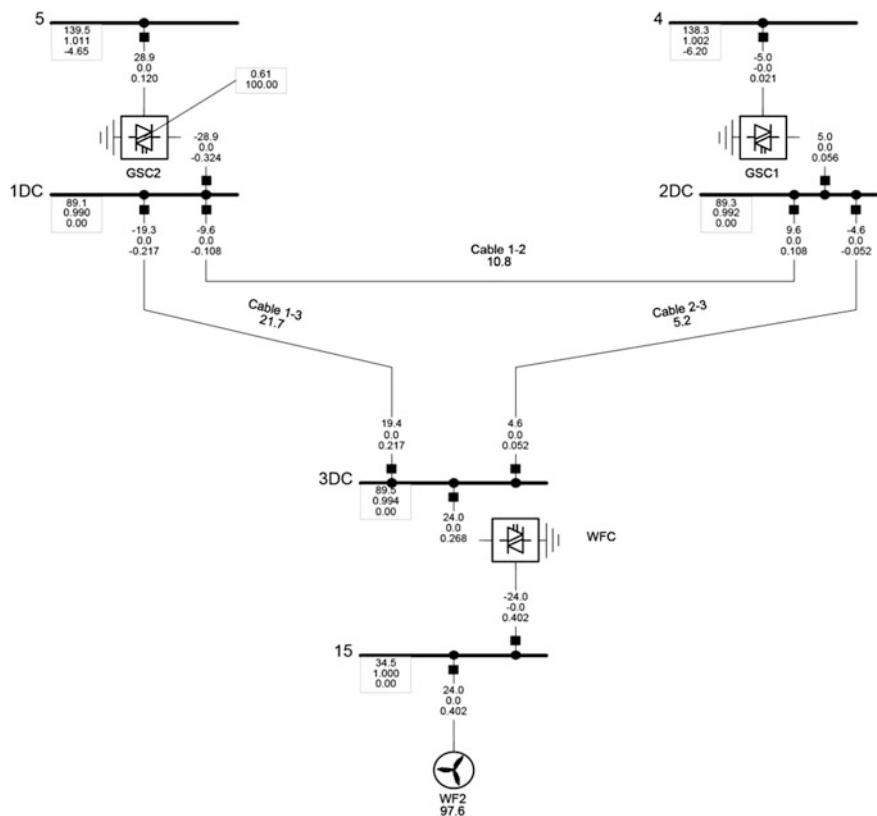
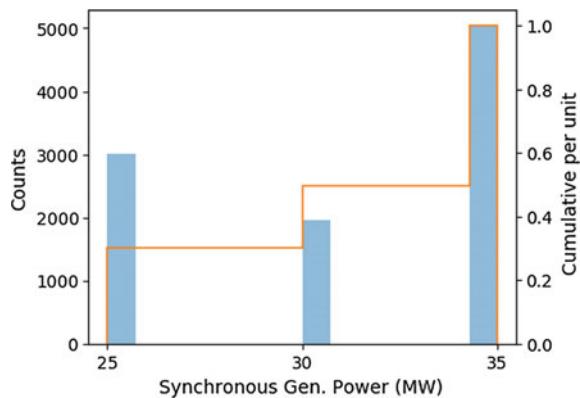
**Fig. 5.25** Three-terminal HVDC transmission system used to integrate the wind power plants (WF1 and WF2)

Fig. 5.26 Histogram for synthetic data of G1.
Three-state empirical distribution: 25, 30 and 35 MW



A MATLAB program has been used to generate the simulation scenarios considered in this chapter; the program used the well-known inverse transform; it produces samples of a desired probability distribution.

Figure 5.26 shows histograms and the discrete cumulative distribution function (CDF) in percentage representative of the simulation scenarios for the active network element. The histograms have been created using the 10,000 samples created using the specific probability distribution describing the active power of the active network elements (Fig. 5.27).

The variability of the power demands is simulated using a normal (or Gaussian) distribution; the continuous distribution is characterized by two parameters, the mean or expectation of the distribution (μ) and the standard deviation (σ). Details of the Gaussian distribution parameters used for the active power demand in each load are presented in Table 5.3.

5.10 Results

The 10,000 simulation scenarios are prepared and loaded into a Microsoft Excel file named “*StochData.xlsx*” (4.83 MB). The script named “**ProbabilisticLF**”, the main program of the PLF implementation, is called inside DIgSILENT PowerFactory. The main DPL script runs for approximately 65.3 s, and a Microsoft Excel file containing the simulations results of the 10,000 scenarios is produced. The MS Excel file named “*StochResults.xlsx*” contains the simulation results of the bus voltages, loading percentage of the transmission lines, active power losses in each transmission line, transmission lines currents (I_{ij}) and power flows (P_{ij}). The output Microsoft Excel file contains the raw data that should be post-processed in order to create the appropriate probabilistic representation of the variables.

The packages pandas [20] and matplotlib [21] are used in the following to process the generated results. The histogram and empirical cumulative distribution

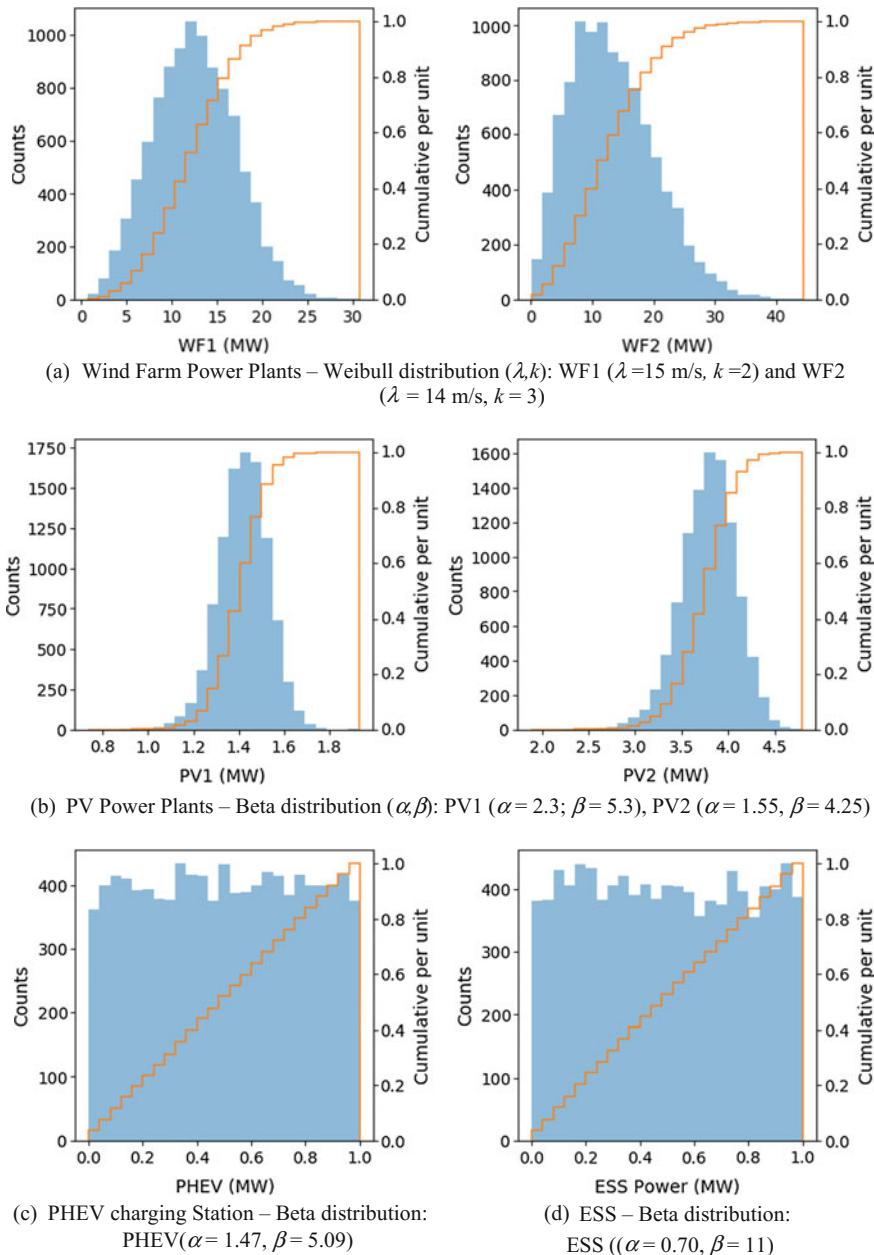


Fig. 5.27 Histograms of synthetic data of all non-CG plants

function are created for the main variables (bus voltages, active power flows, currents and loading percentage of transmission lines and total system power losses). Only the most relevant results are shown here because of space constraints.

Figure 5.28 shows for each bus, the maximum, mean and minimum voltages for the total scenarios are covered in the study. It can be noticed that some buses have an effective voltage control which made this magnitude insensitive to load and generation variations (buses 1 and 15). Other voltages deeply vary depending on the conditions of the considered scenario. The lowest voltage is reached at bus 3 with a value of 0.924 pu, while the highest voltage for all scenarios happens at bus 14 being 1.061 pu.

Figure 5.29 illustrates statistical information concerning the voltage at two loading buses. It is noticed that at bus 14, the range of variation is quite wide, fluctuating from almost 0.93 to 1.06 pu. Instead, at bus 4, the range of voltage variation is considerably narrower, going from 0.99 to 0.935 pu. This is also perceived looking at the σ values indicated for each bus. The wider variation for voltage 14 can be explained by the presence of a wind generator on this bus with a strong variable power injection.

It is worth to mention that the statistical distributions are not following a pure normal (or Gaussian) curve, basically because the different probabilistic distribution functions (three-states, beta, Weibull, normal) used to represent the variability of the active and passive network components (Figs. 5.30 and 5.31).

In Fig. 5.32, the power flows through all the lines and cables, in MW, are depicted providing the maximum, mean and minimum values for all the scenarios. The lines with wider variation are more influenced by the variability of uncontrollable power injections (PV, PHEV, WF). This is noticeable in lines 9–14 and 13–14 which must transfer the variable generation of generator WF1. The histograms, medians and standard deviations of power flows across those lines are shown in Fig. 5.33. As expected, the distributions in both pictures are similar and related to the active power injection by WF1. In fact, when the power injection on

Table 5.3 Details of the Gaussian distribution parameters used for modelling the load demand

Load	Bus	Mean active power (μ in MW)	Standard deviation (% of the μ)
L2	2	21.95	1
L3	3	102.22	25
L4	4	45.14	20
L5	5	5.80	7
L6	6	11.39	15
L9	9	27.34	5
L10	11	4.35	10
L11	11	4.35	2
L12	12	5.88	8
L13	13	12.46	1
L14	14	14.99	4

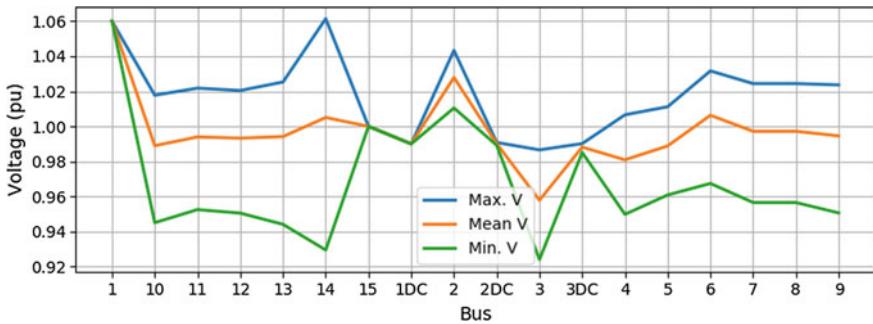


Fig. 5.28 Illustration of voltage variations along the network. The maximum, mean and minimum value at each bus for the 10,000 scenarios are represented

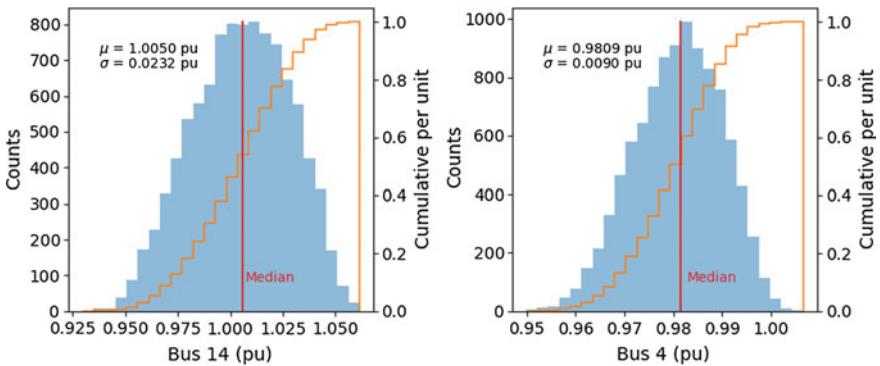


Fig. 5.29 Statistical data related to AC bus voltages. Left: histogram and cumulative distribution of voltage at bus 14. Right: histogram and cumulative distribution at bus 4

bus 14 is low, the power flows from bus 9 to 14 and from 13 to 14 (positive values in Fig. 5.33).

Figure 5.33 displays what happens in the DC part of the system. Here, there are no loads, and the cables are designed to evacuate the power generated by the offshore WF2. This is the reason why power flows are always positive across cable 1–3.

Figure 5.34 depicts the loading of the system branches in all the studied scenarios. It can be noticed that there are no overloads in any scenario and that the most loaded lines are those connecting buses with synchronous generators (line 1–2, line 1–5, line 2–3) (Fig. 5.35).

Finally, Fig. 5.36 shows the histograms of active power losses (MW) and cumulative distribution for the whole system. The mean value is 12.6 MW, but in a few scenarios, the power losses could rise to almost 18 MW, mainly influenced by the generation variability and the path followed by the current to reach the loads.

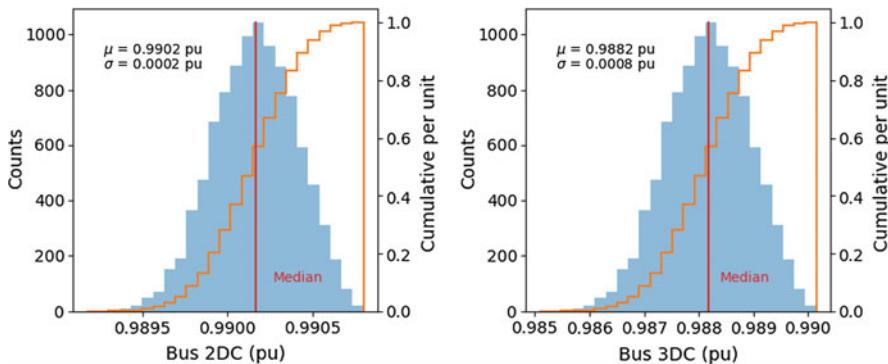


Fig. 5.30 Statistical data related to bus voltage at DC buses. Left: histogram and cumulative distribution of voltage at bus 2DC. Right: histogram and cumulative distribution at bus 3DC

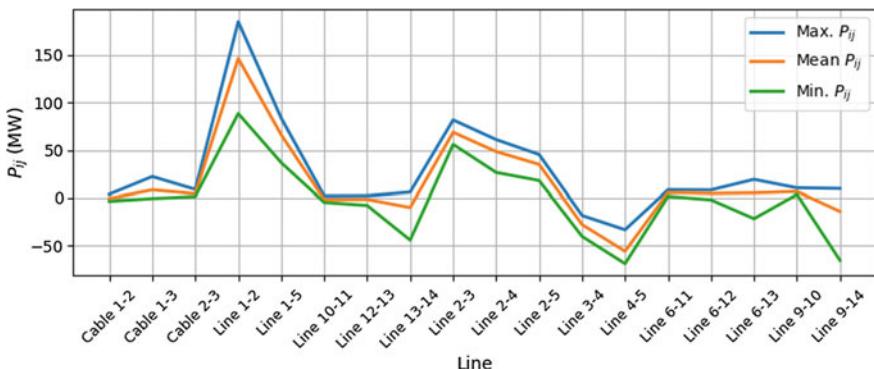


Fig. 5.31 Illustration of lines power flow variations. The maximum, mean and minimum value at each line for the 10,000 scenarios are represented

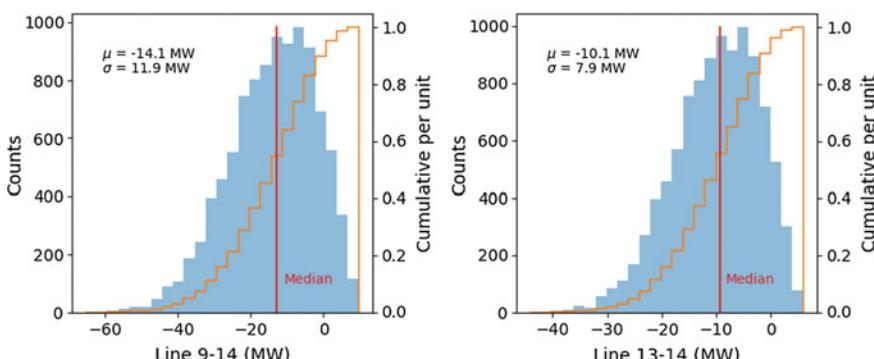


Fig. 5.32 Statistical data related to AC power flows. Left: histogram and cumulative distribution of active power flow through line 9-14 (9 is the sending terminal). Right: histogram and cumulative distribution of active power flow through line 13-14 (13 is sending terminal)

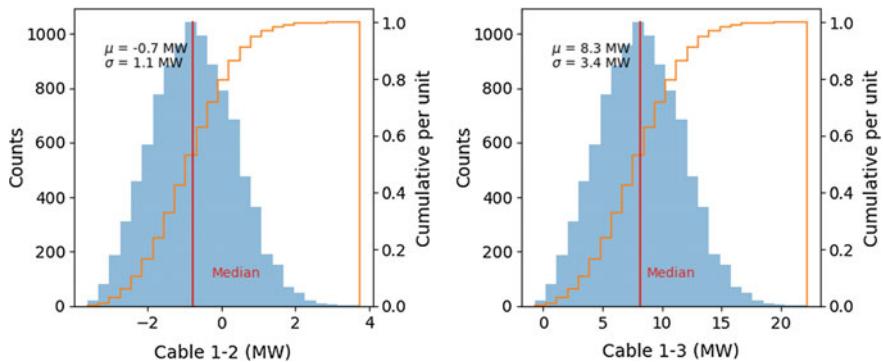


Fig. 5.33 Statistical data related to DC power flows. Left: histogram and cumulative distribution of active power flow through cable 1–2. Right: histogram and cumulative distribution of active power flow through cable 1–3

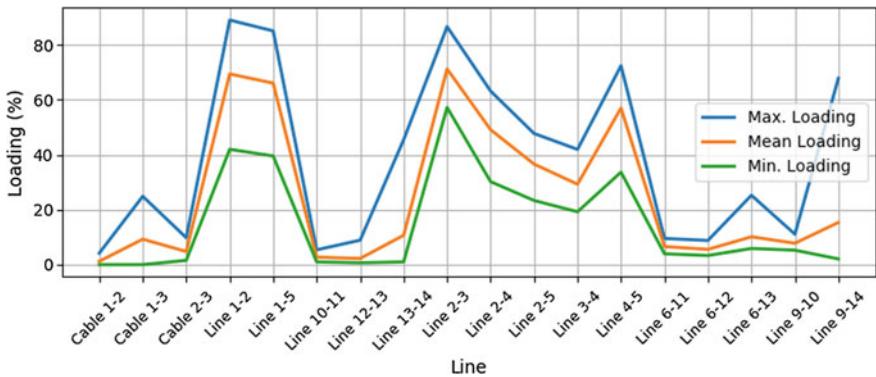


Fig. 5.34 Percentage of loading of system branches. Maximum, mean and minimum values for the 10,000 studied cases are represented

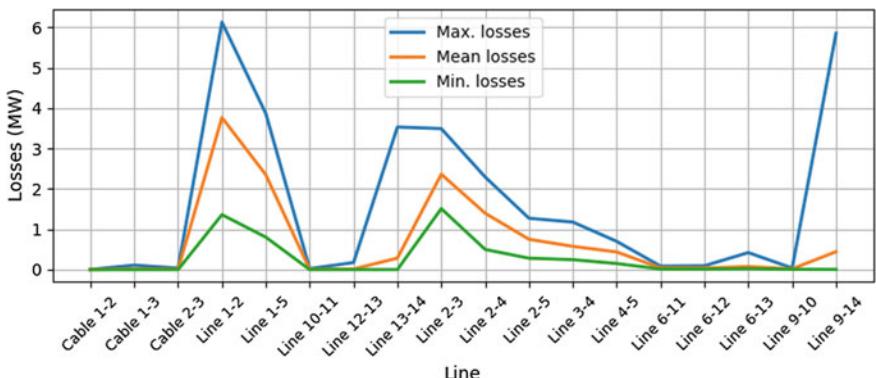


Fig. 5.35 Maximum, mean and minimum losses values (MW) in all branches for the 10,000 studied scenarios

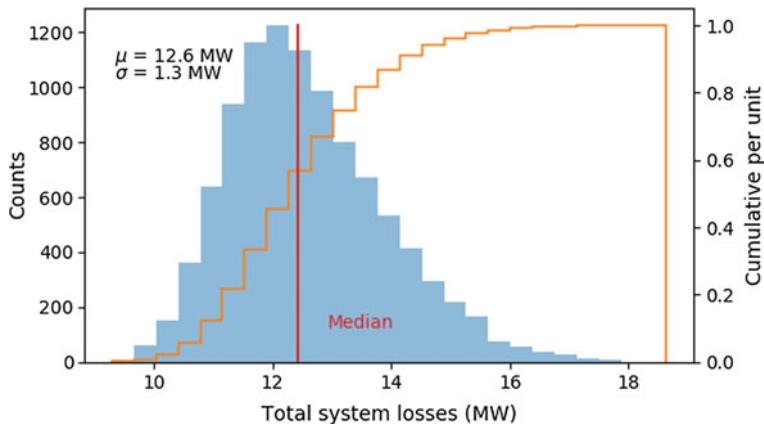


Fig. 5.36 Illustrative histograms of total system active power losses (MW) including cumulative distribution

5.11 Conclusions

Deterministic load-flow analysis is an effective tool that is commonly used to capture the power system operational performance and its state at a certain point of time. However, modern power systems are characterized by an increasing penetration of new technologies adding uncertainties to the design and operation. This chapter presents the *DIGSILENT PowerFactory script language* (DPL) implementation of a DPL script to perform probabilistic power flow (PLF) using MCS in order to consider the variability of the stochastic variables in the power system during the assessment of the steady-state performance. The proposed DPL uses as input data the stochastic data coming from an external Microsoft Excel file, and then, the DPL is able to carry on a probabilistic load-flow and exports the results using a Microsoft Excel file. A modular programming approach has been used to provide flexibility and portability of the script. Internal matrices (*IntMatrix*) are used to transfer data between the subscripts; this approach allows to re-use the code in some other application. The suitability of the implemented DPL is illustrated using the classical IEEE 14 buses.

References

1. F. Alvarado, Y. Hu, R. Adapa, Uncertainty in power system modeling and computation, in *Proceedings, 1992 IEEE International Conference on Systems, Man, and Cybernetics*, vol. 1, pp. 754–760 (1992)
2. P. Chen, Z. Chen, B. Bak-Jensen, Probabilistic load flow: a review, in *Third International Conference on Electric Utility Deregulation and Restructuring and Power Technologies, 2008. DRPT 2008*. pp. 1586–1591 (2008)

3. G.J. Anders, *Probability Concepts in Electric Power Systems* (Wiley-Interscience, New York, 1990)
4. J. Mur-Amada, A.A. Bayod-Rujula, Wind power variability model Part II—probabilistic power flow, in *9th International Conference on Electrical Power Quality and Utilisation, 2007. EPQU 2007*, pp. 1–6 (2007)
5. B. Borkowska, Probabilistic load flow. *IEEE Trans. Power Apparatus Syst.* **PAS-93**(3), 752–759 (1974)
6. B.R. Prusty, D. Jena, A critical review on probabilistic load flow studies in uncertainty constrained power systems with photovoltaic generation and a new approach. *Renew. Sustain. Energy Rev.* **69**, 1286–1302 (2017)
7. R.N. Allan, B. Borkowska, C.H. Grigg, Probabilistic analysis of power flows, in *Proceedings of the Institution of Electrical Engineers*, vol. 121, no. 12, pp. 1551–1556 (1974)
8. A.M. Leite da Silva, S.M.P. Ribeiro, V.L. Arenti, R.N. Allan, M.B. Do Couto Filho, Probabilistic load flow techniques applied to power system expansion planning. *IEEE Trans. Power Syst.* **5**(4), 1047–1053 (1990)
9. F. Milano, *Power system modelling and scripting*, 1st edn. (Springer, New York, 2010)
10. G.W. Stagg, A.H. El-Abiad, *Computer Methods in Power System Analysis* (McGraw-Hill series in electronic systems) (McGraw-Hill, New York, 1968), 427 p
11. H.E. Brown, *Solution of Large Networks by Matrix Methods*, 2nd edn. (Wiley, New York, Chichester, 1985), pp. xv, 320 p
12. J. Arribalzaga, C.P. Arnold, *Computer Analysis of Power Systems* (Wiley, Chichester, England, New York, 1990), pp. xiii, 361 p
13. T.J. Williams, C. Crawford, Probabilistic power flow modeling: renewable energy and PEV grid interactions, presented at the The Canadian Society for Mechanical Engineering Forum 2010, CSME FORUM 2010, Victoria, British Columbia, Canada, 7–9 June 2010
14. Z. Pei, S.T. Lee, Probabilistic load flow computation using the method of combined cumulants and Gram-Charlier expansion. *IEEE Trans. Power Syst.* **19**(1), 676–682 (2004)
15. Z. Hu, W. Xifan, A probabilistic load flow method considering branch outages. *IEEE Trans. Power Syst.* **21**(2), 507–514 (2006)
16. F. Coroiu, C. Velicescu, C. Barbulescu, Probabilistic and deterministic load flows methods in power systems reliability estimation, in *EUROCON—International Conference on Computer as a Tool (EUROCON), 2011 IEEE*, pp. 1–4 (2011)
17. M. Aien, R. Ramezani, S.M. Ghavami, Probabilistic load flow considering wind generation uncertainty. *ETASR Eng. Technol. Appl. Sci. Res.* **1**(4), 126–132 (2011)
18. P. Jorgensen, J.S. Christensen, J.O. Tande, Probabilistic load flow calculation using Monte Carlo techniques for distribution network with wind turbines, in *8th International Conference on Harmonics And Quality of Power, 1998. Proceedings*, vol. 2, pp. 1146–1151 (1998)
19. *DIGSILENT PowerFactory 2016—User Manual* (DIGSILENT GmbH, Gomaringen, 2016)
20. W. McKinney, Data structures for statistical computing in python, in *Proceedings of the 9th Python in Science Conference*, pp. 51–56 (2010)
21. J.D. Hunter, Matplotlib: a 2D graphics environment. *Comput. Sci. Eng.* **9**, 90–95 (2007)

Chapter 6

Dynamic Stability Improvement of Islanded Power Plant by Smart Power Management System: Implementation of PMS Logic



Hamid Khoshkho and Ali Parizad

Abstract Power management system (PMS) which is a kind of special protection system is used to detect predefined conditions and execute timely remedial actions to prevent instability or improve operating point, especially in an islanded system. To this purpose, PMS usually includes several functions such as load shedding/sharing, generation shedding, and generation mode control which are automatically executed to improve operating point. In this chapter, to show the capability of DIgSILENT power factory to simulate smart grids functionalities, DIgSILENT programming language (DPL) is used to model PMS logic in automatically detecting islanding condition as well as executing load/generation shedding in an islanded system to prevent instability. Indeed, this modelling gives the possibility to check the impact of considered PMS logic under different operating conditions on the stability of the system.

Keywords DIgSILENT power factory · DPL · DIgSILENT simulation language (DSL) · Load/generation shedding · Power management system (PMS)
Special protection system

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_6) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

H. Khoshkho (✉)

Department of Electrical Engineering, Sahand University of Technology,
Sahand New Town, Tabriz, Iran
e-mail: khoshkho@sut.ac.ir

A. Parizad

MAPNA Electrical & Control Company, Tehran, Iran
e-mail: ali.parizad@ieee.org

6.1 Introduction

During the last decade, several system instabilities have occurred in power systems all over the world which result in huge economic losses. In this regard, extensive investigations have been carried out to provide proper tools to assist system operators in timely executing remedial actions to prevent instability. PMS is an intelligent centralised system which measures required variables from different substations, performs required analysis (such as power flow), calculates intended variables (to detect predefined or abnormal conditions), and executes timely control actions [1–3]. Especially, after an unintentional islanding occurrence, which usually needs timely reaction of operators, PMS can play an important role to automatically detect islanding condition and perform proper remedial actions in a fraction of a second to regain operating point to the acceptable region. In this condition, to balance load and generation, PMS usually executes either load or generation shedding procedure, and then, it can execute secondary frequency control to precisely regulate the system frequency [1, 4, 5]. In addition to automatically execution of remedial actions, PMS provides the possibility for operators to monitor dynamic behaviour of the system and execute intended control actions.

Since PMS usually executes automatic control actions to improve the operating point, performing accurate offline simulations to assess the effectiveness of considered PMS logic is inevitable. Reviewing the literature shows that although papers discuss the impact of the PMS on the stability of systems, they have not explained the implementation of PMS logic in a simulation software and also have not discussed the detailed logic of PMS. This chapter aims to show the ability of DIgSILENT power factory in testing a PMS logic by simulating the execution of automatic remedial actions. In this regard, DPL which is a proper tool to simulate PMS logic is used to automatically detect system status, make a decision according to considered PMS logic and perform the selected control action(s).

Among different functions used in PMS logic, in this chapter, the load shedding, generation shedding, and secondary frequency control procedures of PMS have been explained in Sect. 6.2. Then, in Sect. 6.3, DPL is used to model PMS logic in performing load/generation shedding and frequency regulation. Finally, simulations results and conclusion have been presented in Sects. 6.4 and 6.5.

6.2 Functions of the PMS

If active power generation of a system is more/less than load demand, the frequency increases/decreases which may result in operation of frequency relays and system black out. Also, some loads, especially in industrial systems, are highly sensitive to frequency deviation and any under/over frequency may cause process failure and economic loss. Therefore, to precisely regulate frequency after an abrupt disconnection of the power plant from the grid, PMS automatically measures the amount

of active power generation and load demand, calculates the amount of load/generation which should be shed, and executes load/generation shedding according to the priority list of loads and generators. In this chapter, among several PMS functions usually used to control the operating point, load shedding, generation shedding, and secondary frequency control procedures are implemented in DPL. In the following subsections, these procedures are introduced, briefly.

6.2.1 Load Shedding Procedure

When islanding condition occurs, if load demand exceeds active power generation, the amount of load which should be shed is calculated as follows and then a set of loads is selected according to priority list to shed:

$$P_{\text{To Be Shed}} = P_{\text{transfer}} - N_G \times SPRes + P_{\text{Offset}} \quad (1)$$

where P_{transfer} is the active power imported from the external grid before islanding condition, N_G is the number of the generator in running mode, $SPRes$ is the active power reserve of each generator, and P_{Offset} is the extra load which should be shed to guarantee stability (in this chapter it is assumed that $P_{\text{Offset}} = 0$).

6.2.2 Generation Shedding Procedure

When islanding condition occurs, this procedure identifies the amount of excessive generation in the islanded system, selects a set of generation units according to a priority list, and sheds this set to prevent over frequency rapidly. Thus, the amount of generation which should be shed may be calculated as follows:

$$P_{\text{To Be Shed}} = P_{\text{transfer}} \quad (2)$$

where P_{transfer} is the active power exported to the external grid before islanding condition. It is worth mentioning that if load operation with speed control (LOSC) mode exists in governor logic [6], fewer generators can be shed. In this situation, the operating mode of other generators should change from load operation with load control (LOLC) to LOSC mode to decrease power generation and prevent significant frequency deviation, quickly. Indeed, in this condition, offline simulations are needed to calculate the exact amount of generation which should be shed (In this chapter 25% of P_{transfer} is shed, and operating mode of other generators are changed to LOSC.).

6.2.3 Secondary Frequency Control

Secondary frequency control is used to precisely regulate the frequency of system by changing the set point of governors. In this chapter, the set points of governors are changes with the step of $dw = 0.005$ p.u. to regulate frequency. Indeed, a proper value for dw should be obtained using offline simulations, and it should be small enough to prevent the oscillatory behaviour.

6.3 Application of DPL in PMS Modelling

DIgSILENT programming language is a proper tool to implement PMS logic and simulate its operation. To show this capability, in the test system shown in Fig. 6.1, DPL is used to simulate the load and generation shedding procedures. In this study, assuming generators and governors of test system are named G_i and pcu_i ; $i = 1, \dots, 5$, and one of the parallel transformers (i.e. $Tr230/400-2$) is out of service, two scenarios will be considered:

- When the active power imports from the grid to the plant, the second transformer ($Tr230/400-1$) is tripped by a protection relay which causes the frequency of plant to decrease. In this situation, a proper and timely load shedding procedure should be employed by PMS to balance load and generation.

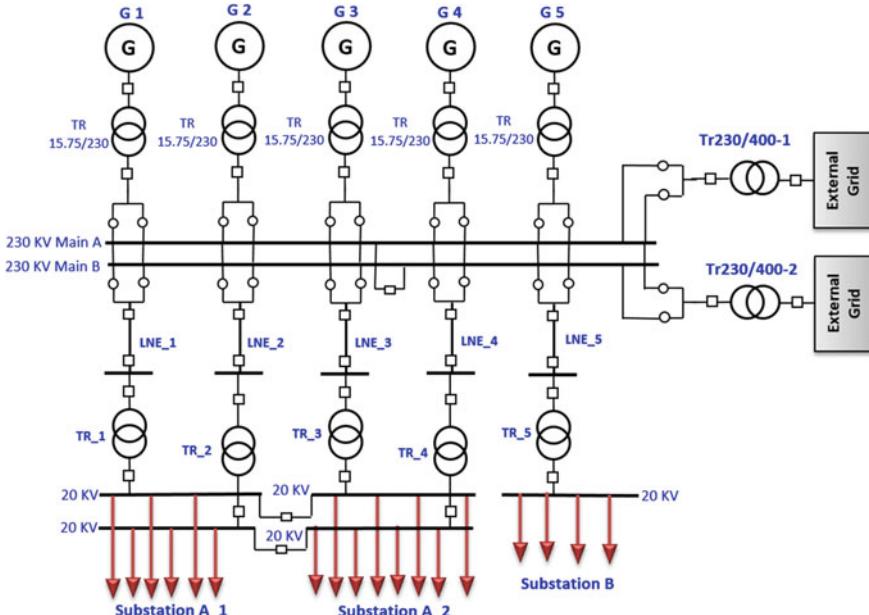


Fig. 6.1 Single line diagram of the test system

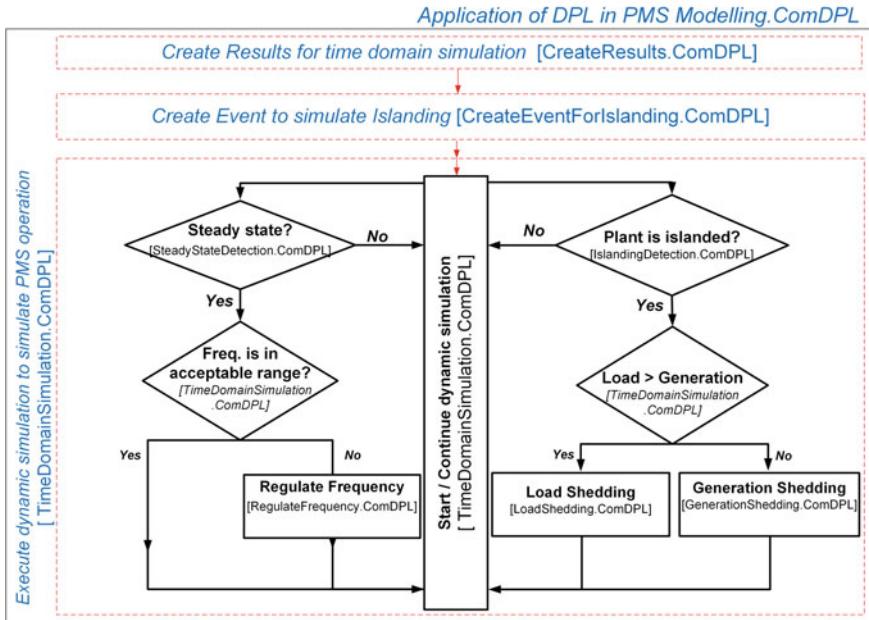


Fig. 6.2 Flow chart of the logic used to simulate PMS operation

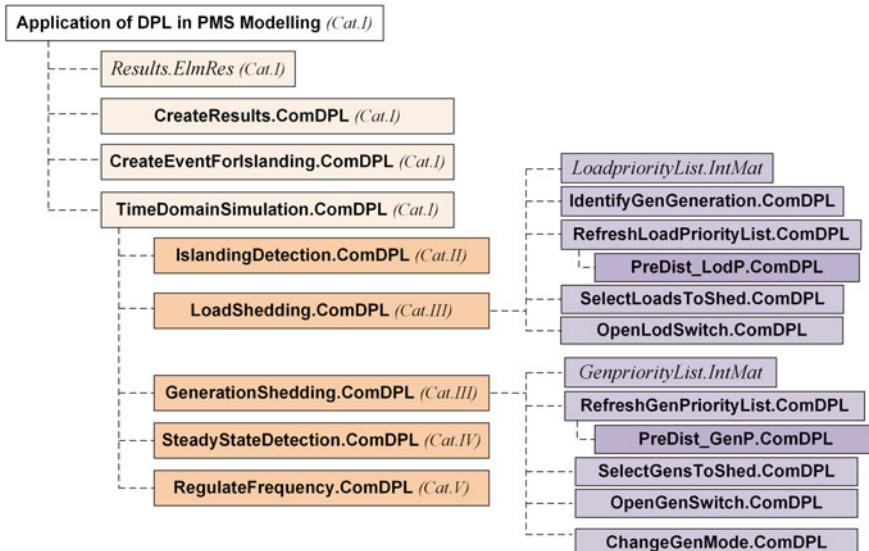


Fig. 6.3 Hierarchical structure of the program

- When the power exports from the plant to the grid, *Tr230/400-1* is tripped. In this condition, execution of timely generation shedding procedure by PMS is necessary to prevent over frequency.

Figure 6.2 shows the flowchart used to model DPL logic in detecting system condition and executing load or generation shedding procedure. Also, the hierarchical structure of DPL scripts prepared for this study is shown in Fig. 6.3.

Scripts in Fig. 6.3 may be classified into following categories:

- **Time-domain simulation to determine the system condition and analysing the network frequency (Cat.I):** This includes *CreateResults.ComDPL*, *CreateEventForIslanding.ComDPL*, and *TimeDomainSimulation.ComDPL*.
- **Detecting the islanding condition to decide the load or generation shedding (Cat.II):** This includes *IslandingDetection.ComDPL* which detects islanding condition occurrence and trigger load or generation shedding procedure.
- **Implementing the load or the generation shedding (Cat.III):** This includes *LoadShedding.ComDPL*, *GenerationShedding.ComDPL*, and their subscripts.
- **Detecting the steady-state condition to check the need to precisely regulate frequency (Cat.IV):** This category includes *SteadyStateDetection.ComDPL* which detects steady-state condition.
- **Secondary frequency control in steady-state condition (Cat.V):** This category includes *RegulateFrequency.ComDPL* which changes governor settings to precisely regulate frequency in steady-state condition.

The contents of the main script, i.e. *Application of DPL in PMS Modelling.ComDPL* are shown in Fig. 6.4.

The main script (*Application of DPL in PMS Modelling.ComDPL*) includes:

- ***Results.ElmRes*:** it is where the results of the time-domain simulation are stored. It should be noted that conventionally, simulation results are stored in “All calculations.ElmRes”. Nevertheless, in this study, *Results.ElmRes* is created and used to store calculation results.
- ***CreateResults.ComDPL*:** it creates desired simulation results in *Results.ElmRes* (see Sect. 6.3.1). These results are then used during time-domain simulation to monitor desired variables such as the active power consumed by loads, the active power exchange with the external grid (passes through *Tr230/400-1*), and the frequency of the plant.
- ***CreateEventForIslanding.ComDPL*:** it creates the event related to an outage of *Tr230/400-1* at $t = 69.5$ s (see Sect. 6.3.2). It should be stated that this event results in islanding condition.

```

!variable declaration
string GridTrns1; !1
int tevent; !2
object Result; !3
double Tstop; !4
ResetCalculation(); !Reset Calculations !5
ClearOutput(); !Clear output window !6
Result = Results; !object "Result" points to Results.ElmRes !7
GridTrns1='Tr230/400-1'; !Transformer 'Tr230/400-1' connects plant to Grid !8
tevent=69.5; !At t=tevent, 'Tr230/400-1' will be disconnected !9
CreateResults.Execute(Result,GridTrns1); !Create results of simulation !10
CreateEventForIslanding.Execute(GridTrns1,tevent); !Create Event for simulation !11
TimeDomainSimulation.Execute(Result,GridTrns1); !Execute Time domainsimulation !12

```

Fig. 6.4 DPL script of the *Application of DPL in PMS Modelling.ComDPL*

- ***TimeDomainSimulation.ComDPL***: it executes dynamic simulation and models PMS functions to prevent over/under frequency by generation/load shedding or to precisely regulate frequency in steady-state condition (see Sect. 6.3.3).

The detailed description of mentioned scripts is explained in Sects. 6.3.1–6.3.3 and simulation results have been presented in Sect. 6.4.

6.3.1 *CreateResults.ComDPL*

Before execution of time-domain simulation, the desired results which may be used during simulation or those which need to be obtained as a result of simulation should be defined. Thus, on line 11 of the main script (see Fig. 6.4) *CreateResults.ComDPL* is executed whose inputs are:

- *Result*: it points to calculation results (line 8 in Fig. 6.4),
- *GridTrns1*: it is *Tr230/400-1* (line 9 in Fig. 6.4). As mentioned before, the trip of this transformer results in islanding condition.

As shown in Fig. 6.5, to save only those variables which are needed, old results are deleted on lines 16–17. Then, a *for* loop (lines 19–30) is used to add the active power of loads ('*m:P:bus1*' in line 23) and the frequency of the load buses ('*m:fe*' in line 29) to *Result* using *AddVar* function. Also, on lines 32–37, the active powers of generators ('*m:P:bus1*' in line 36) are added to *Result*. Moreover, on lines 39–43, the active power imports from (or exports to) the grid, which passes through HV side of *Tr230/400-1.ElmTr2*, is added to *Result* ('*m:P:bushv*' in line 43). Finally, on lines 45–54, governor settings ('*s:wref2*' in line 52 which will be used to regulate the active power of generation unit) are added to *Result*.

6.3.2 *CreateEventForIslanding.ComDPL*

In addition to results, intended event(s) should be defined before execution of time-domain simulation. Thus, on line 12 of Fig. 6.4, *CreateEventForIslanding.ComDPL* is executed to define the outage of *Tr230/400-1.ElmTr2* at $t = 69.5$ s as an event which results in islanding condition. As shown in Fig. 6.4, *GridTrns1* (which is *Tr230/400-1*) and *tevent* ($=69.5$ s) are applied to *CreateEventForIslanding.ComDPL* as input. Figure 6.6 shows *CreateEventForIslanding.ComDPL* in which all old events are assigned to *sEvent* set on line 10 to be deleted on line 11. Then, to create the above-mentioned event (i.e. outage of *Tr230/400-1* at $t = 69.5$ s), the transformer *Tr230/400-1* is assigned to *OTr* on line 15, and a switch event related to this transformer is created on lines 16 and 17. The time of this switch event is set to *tevent* on line 18 which results in transformer outage at $t = 69.5$ s.

```

!variable declaration                                !1
int EndCnd,cmp,i;                                !2
object mon,O,Obus,Olod,OTr,OSym,Inc,EventDir,ODsl; !3
string str1,str2;                                !4
set oldmon,sLod,allBars,sTr,sSym,sDsl;           !5
!Create a set of all Busbars,transformers,loads,generators and DSLs... !6
! and sort them to their names (from A to Z)      !7
allBars = AllRelevant('.StaBar,*.ElmTerm'); allBars.SortToName(0);    !8
sTr   = AllRelevant('.ElmTr2');                   sTr.SortToName(0);     !9
sLod  = AllRelevant('.ElmLod');                   sLod.SortToName(0);    !10
sSym   = AllRelevant('.ElmSym');                  sSym.SortToName(0);    !11
sDsl  = AllRelevant('.ElmDsl');                  sDsl.SortToName(0);    !12
Inc   = GetCaseCommand('ComInc');                 !13
EventDir= Inc:p_event;                           !14
! Get old contents of results (from previous simulations) and Delete them !15
oldmon = Result.GetContents();                   !16
Delete(oldmon);                                !17
!Active power of loads and frequency of load buses are added to results !18
for(Olod=sLod.First();Olod;Olod=sLod.Next()){
    str1=Olod:loc_name;
    mon=Result.CreateObject('IntMon', str1);
    mon:obj_id = Olod;
    mon.AddVar('m:P:bus1');    ! Add the Active Power (P) of load to results. !23
    O=Olod:bus1;              !24
    Obus=O:cBusBar;          !25
    str2=Obus:loc_name;
    mon=Result.CreateObject('IntMon', str2);
    mon:obj_id = Obus;
    mon.AddVar('m:fe');       !Add the frequency of load busesto results. !29
}
!Active power of Generators are added to results      !31
for(OSym=sSym.First();OSym;OSym=sSym.Next()){
    str1=OSym:loc_name;
    mon=Result.CreateObject('IntMon', str1);
    mon:obj_id = OSym;
    mon.AddVar('m:P:bus1');
}
!Active power passes through transformer 'Tr230/400-1' is added to results !38
str1=sprintf('%s%s',GridTrns1,'.ElmTr2');      !str1='Tr230/400-1.ElmTr2'
OTr = sTr.FirstFilt(str1);
mon=Result.CreateObject('IntMon', str1);
mon:obj_id = OTr;
mon.AddVar('m:P:bushv');
!'wref2' of governors are added to results        !44
for (ODsl=sDsl.First();ODsl;ODsl=sDsl.Next()){
    str1 = ODsl:loc_name;
    i = strstr(str1, 'pcu');
    if({i>=0}){
        str2=sprintf('%s%s',str1,'.ElmDsl');
        mon=Result.CreateObject('IntMon', str2);
        mon:obj_id = ODsl;
        mon.AddVar('s:wref2');
    }
}

```

Fig. 6.5 DPL script of the *CreateResults.ComDPL*

6.3.3 TimeDomainSimulation.ComDPL

After creating intended results and event, *TimeDomainSimulation.ComDPL* is executed on line 13 of the main script (see Fig. 6.4). As shown in this figure, the inputs of *TimeDomainSimulation.ComDPL* are:

```

!variable declaration                                !1
string str;                                         !2
object OTr,Inc,EventDir,event;                      !3
set sTr,sEvent;                                     !4
!Create a set of all transformers and sort them to their names (from A to Z) !5
sTr = AllRelevant('*_ElmTr2');      sTr.SortToName(0);           !6
! Delete Existing Event Set                         !7
Inc      = GetCaseCommand('ComInc');                !8
EventDir = Inc:p_event;                           !9
sEvent   = EventDir.GetContents();                 !10
Delete(sEvent);                                    !11
EchoOff ();                                       !12
! Create New Switch Event for 'Tr230/400-1.ElmTr2' !13
str=printf("%s%s",GridTrns1,'.ElmTr2');      !str='Tr230/400-1.ElmTr2' !14
OTr = sTr.FirstFilt(str);                        !15
event = EventDir.CreateObject('EvtSwitch','Switch Event'); !16
event:p_target = OTr;                            !17
event:time     = tevent;                          !18

```

Fig. 6.6 DPL script of the *CreateEventForIslanding.ComDPL*

- *Result*: this object points to *Results.ElmRes* where the results are stored.
- *GridTrns1*: this string contains *Tr230/400-1* (see line 9 in Fig. 6.4). The outage of this transformer at $t = 69.5$ s causes the plant to be islanded.

The main goal of *TimeDomainSimulation.ComDPL* is to simulate the outage of *Tr230/400-1* at $t = 69.5$ s and the operation of PMS logic. In this script, just after the islanding condition occurrence, load shedding (*LoadShedding.ComDPL*), or generation shedding (*GenerationShedding.ComDPL*) procedure will be executed to prevent under or over frequency, respectively. Also, when the system reaches to steady-state condition, *RegulateFrequency.ComDPL* may be executed automatically to precisely regulate system frequency.

According to Fig. 6.7, in *TimeDomainSimulation.ComDPL* which includes six subscripts, a *do-while* loop (lines 18-47) is used to perform the following commands, iteratively:

- Stop time of simulation is set to *Tstop* on line 19 (initially *Tstop* = 65 s). In the next iterations, *Tstop* increases by 1 s (line 46 in Fig. 6.7) and time-domain simulation will continue.
- Time-domain simulation is executed until *Tstop* (line 20).
- *IslandingDetection.ComDPL* is executed every $dT = 1$ s (line 23) to identify whether the plant is islanded or not. Obviously, this means that PMS assess islanding occurrence every second. If this script identifies islanding condition, the islanding parameter, i.e. *Islanding*, and *MWtransfer* parameters, will be set to 1 and 0 on lines 24 and 25, respectively. Otherwise, *Islanding* will be set to 0 (line 24) and the active power passes through *Tr230/400-1.ElmTr2* will be assigned to *MWtransfer* (line 25).

It should be noted that *MWtransfer* shows the active power imports from/exports to the grid at the moment that *IslandingDetection.ComDPL* is executed (once a second), but *Ptransfer* (line 27 in Fig. 6.7) indicates the active power imports from(exports to the grid just before islanding occurrence. In this regard,

```

!variable declaration                                !1
int LodShedNeeded,GenShedNeeded,Islanding,SteadyState;      !2
double Tstop,dT,Ptransfer,MWtransfer,favr,TRF,TLS;        !3
object Ldf,Inc,Sim;                                         !4
!Call power flow, Initializing, and time domain simulation analyses   !5
Ldf    = GetCaseCommand('ComLdf');                         !6
Inc    = GetCaseCommand('ComInc');                          !7
Sim    = GetCaseCommand('ComSim');                         !8
Echo.Off();                                                 !9
Ldf.Execute();      !Calculate Load Flow Analysis          !10
Inc.Execute();     !Calculate initialize Condition       !11
TRF=0;           !the last time at which RegulateFrequency.ComDPL was executed !12
Tstop=65;         !Initial time domain simulation time       !13
dT=1;            !'dT' is step time to check islanding        !14
LodShedNeeded=1;  !"LodShedNeeded=1": activate load shedding procedure !15
GenShedNeeded=1;  !"GenShedNeeded=1": activate load shedding procedure !16
Islanding=0;      !"Islanding =1": means that the plant is disconnected from grid !17
do{
    Sim:tstop = Tstop;        !Stop time of time domain simulation set to Tstop !19
    Sim.Execute();           !Run Time Domain Simulation until Tstop          !20
    LoadResData(Result);    !Load results data                           !21
!Identify whether the plant is Islanded               !22
    IslandingDetection.Execute(GridTrns1);                !23
    Islanding = IslandingDetection:Islanding;             !24
    MWtransfer = IslandingDetection:MWtransfer;           !25
    if(Islanding<0){
        Ptransfer=MWtransfer;!active power passes through GridTrns1 before islanding !27
    }
!Execute Load shedding or Generation Shedding after Islanding !29
    if ({Islanding}.and.{LodShedNeeded=1}.and.{Ptransfer>0}){
        LoadShedding.Execute(Tstop,dT,Ptransfer,Result);
        LodShedNeeded=LoadShedding:LodShedNeeded;
    }elseif({Islanding}.and.{GenShedNeeded=1}.and.{Ptransfer<0}){
        GenerationShedding.Execute(Tstop,dT,Ptransfer,Result);
        GenShedNeeded=GenerationShedding:GenShedNeeded;
    }
!Identify whether operating point reaches to steady state !37
    SteadyStateDetection.Execute(Result,Tstop);
    SteadyState=SteadyStateDetection:SteadyState;!SteadyState=1:steady state cond. !39
    favr=SteadyStateDetection:favr;                      !Average frequency during last 3s !40
!Regulate Frequency in steady state condition        !41
    if({SteadyState=1}.and.{abs(favr-1)>0.005}.and.{Tstop>(TRF+3)}){ !42
        RegulateFrequency.Execute(favr,Tstop,TRF);
        TRF=RegulateFrequency:TRF;                            !43
    }
    Tstop=Tstop+dT;           !increase Tstop (stop time) by dT.           !45
}while(Tstop<200)                                         !46
                                                !47

```

Fig. 6.7 DPL script of the *TimeDomainSimulation.ComDPL*

if the plant is islanded (i.e. $Islanding = 1$), $P_{transfer}$ will not be changed. Otherwise, $P_{transfer}$ is set to $MW_{transfer}$ on line 27. Also, $P_{transfer} > 0$ indicates that the plant imports the active power and $P_{transfer} < 0$ shows that the plant exports the active power to the grid before islanding condition.

- Load shedding procedure (*LoadShedding.ComDPL*) will be executed on line 31, if the load demand is more than the generation ($P_{transfer} > 0$) in islanded plant ($Islanding = 1$) and ($LodShedNeeded = 1$). This script selects and sheds some loads according to the priority list of loads.
- Generation shedding procedure (*GenerationShedding.ComDPL*) will be executed on line 34, if the generation is more than the load demand ($P_{transfer} < 0$)

- in islanded plant ($Islanding = 1$) and ($GenShedNeeded = 1$). This script sheds some generators according to the priority list of generators.
- *SteadyStateDetection.ComDPL* will be executed on line 38 to identify whether the system is in steady-state condition or not. In steady-state condition, *SteadyState* will be set to 1 (line 39) and *favr* will show the average frequency during the last 3 s (line 40). In steady-state condition, *RegulateFrequency.ComDPL* may be executed to precisely regulate the system frequency.
 - If operating point reaches to steady-state condition and the frequency deviation from nominal value becomes more than a pre-specified value (here 0.005 p.u.), then *RegulateFrequency.ComDPL* will be executed on line 43 to regulate frequency.

In Sects. 6.3.3.1–6.3.3.5, subscripts used in *TimeDomainSimulation.ComDPL* have been described.

6.3.3.1 IslandingDetection.ComDPL

This script identifies disconnection of the plant from the grid. Figure 6.7 shows that *GridTrnsI* (=Tr230/400-1) is applied to this script as input (line 23). As shown in Fig. 6.8, on lines 9–10 of *IslandingDetection.ComDPL*, the transformer whose name is *GridTrnsI* (= Tr230/400-1) is found and assigned to *OTr*. Then, those cubicles connected to this transformer are found (line 13) and the corresponding switches status is identified (line 16). In line 17 of this script, *opn* = 1 means that the switches connecting *GridTrnsI* to adjacent buses are open and the plant is islanded. Thus, *Islanding* is set to 1 and *MWtransfer* is set to 0 (lines 18 and 19). Otherwise, the active power passes through *GridTrnsI* will be assigned to *MWtransfer* (line 22). Finally, *Islanding* and *MWtransfer* are sent to *TimeDomainSimulation.ComDPL* script as output (see lines 24 and 25 in Fig. 6.7).

6.3.3.2 LoadShedding.ComDPL

Immediately after islanding detection ($Islanding = 1$), if the generation is less than load demand in islanded plant (i.e. $P_{transfer} > 0$), *LoadShedding.ComDPL* executes to select some loads according to the priority list and then shed the selected loads to balance generation and load (Fig. 6.9). As shown on line 31 of Fig. 6.7, *LoadShedding.ComDPL* inputs are:

- *Tstop*: it indicates the time up to which the simulation has been executed.
- *dT*: it is the step time to assess whether the plant is connected to the grid or not.
- *Ptransfer*: it indicates the active power transfers from the grid to the plant before islanding condition.
- *Result*: it points to *Results.ElmRes* where the results of the simulation are stored.

```

!variable declaration                                !1
string str;                                         !2
object OTr,OCub,OSwitch;                           !3
int iCount,iCub,opn;                               !4
set sTr,sCntnt;                                  !5
!Create a set of all transformers and sort them to their names (A to Z) !6
sTr = AllRelevant('* ElmTr2'); sTr.SortToName(0); !7
!Find the transformer whose name is 'Tr230/400-1' !8
Str = sprintf('%s',GridTrns1,'ElmTr2'); lstr1='Tr230/400-1.ElmTr2' !9
OTr = sTr.FirstFilt(str);                         !OTr points to Tr230/400-1.ElmTr2 !10
iCount = OTr.GetConnectionCount();    !the number of electrical connections !11
for (iCub=0; iCub<iCount; iCub=iCub+1){          !12
    OCub=OTr.GetCubicle(iCub);   !OCub is a cubicle of OTr !13
    sCntnt=OCub.GetContents();   !sCntnt includes contents of OCub !14
    OSwitch=sCntnt.First();     !OSwitch points to switch in OCub !15
    opn = OSwitch.IsOpen();      !opn=1: means that the transformer switch is open !16
    if(opn=1){                  !17
        Islanding=1;           !18
        MWtransfer=0;           !19
    }else{                      !20
        Islanding=0;           !21
        MWtransfer=OTr:m:P:bushv; !22
    }
}

```

Fig. 6.8 DPL script of the *IslandingDetection.ComDPL*

```

! variable declaration                                !1
object EventDir,LoadPriorityList;                 !2
int NG;                                           !3
double SPRes;                                     !4
LoadPriorityList = LoadpriorityList; !LoadPriorityList points to Load priority List !5
!calculate active power generation of generators: !6
IdentifyGenGeneration.Execute();                  !7
NG =IdentifyGenGeneration:NG;                   !8
SPRes=IdentifyGenGeneration:SPRes;             !9
!Refresh Priority List:                          !10
RefreshLoadPriorityList.Execute(Result,LoadPriorityList,Tstop,dT); !11
!select optimal load set to shed according to Priority List: !12
SelectLoadsToShed.Execute(LoadPriorityList,Ptransfer,NG,SPRes); !13
!Execute Load Shedding:                        !14
OpenLodSwitch.Execute(LoadPriorityList,Tstop);    !15
LodShedNeeded=0;                                 !16

```

Fig. 6.9 DPL script of the *LoadShedding.ComDPL*

As shown in Fig. 6.3, *LoadShedding.ComDPL* includes four subscripts and a load priority list (*LoadpriorityList.IntMat*). *IdentifyGenGeneration.ComDPL* is the first subscript, executed on line 7 of *LoadShedding.ComDPL* (Fig. 6.9) to identify the number of generators in service (*NG*) as well as the active power reserve of these generators (*SPRes*). Also, *RefreshLoadPriorityList.ComDPL* is executed on line 11 of Fig. 6.9 to update load priority list which will be then used to select a proper set of loads to shed. Then, the information obtained from *IdentifyGenGeneration.ComDPL* and *RefreshLoadPriorityList.ComDPL* is used by *SelectLoadsToShed.ComDPL* to determine a list of loads (according to the priority list) which should be shed. This list is then applied to *OpenLodSwitch.ComDPL* to shed selected loads. After execution of load shedding, *LodShedNeeded* is set to zero

```

!variable declaration !1
int i; !2
set sSyn; !3
object O; !4
!Create a set of all transformers and sort them to their names (A to Z) !5
sSyn = AllRelevant (*.ElmSym'); sSyn.SortToName(0); !6
NG=0; !7
for (0=sSyn.First();0<sSyn.Next()){ !8
    i=0:outserv; !9
    if(i==0){ !10
        NG=NG+1; !11
    }
}
SPRes=7; !MW !12
!13
!14

```

Fig. 6.10 DPL script of the *IdentifyGenGeneration.ComDPL*

to prevent more load shedding, and it should be set to 1 if the plant connects to the grid, again. In the sequel, the subscripts of *LoadShedding.ComDPL* are explained.

IdentifyGenGeneration.ComDPL:

As mentioned before, this subscript is used to identify the number of generators which are in service and generate active power (*NG* in lines 7–13 of Fig. 6.10) as well as the active power reserve of these generators, *SPRes*. Here, *SPRes* for all generators is assumed to be 7 MW.

RefreshLoadPriorityList.ComDPL:

This script which is shown in Fig. 6.11 is used to update the second and third columns of load priority list (*LoadpriorityList.IntMat*). The structure of *LoadPriorityList* matrix, implemented in DPL, is shown in Fig. 6.12. The first column of this matrix shows the priority of loads. Here, the priority of loads which should not be shed is set to zero and loads whose priority are 1 are the first group of loads selected to be shed. The second column of *LoadPriorityList* matrix shows whether the load is in service or not. Also, the third column of this matrix indicates the active power consumed by each load just before islanding condition. Moreover, the fourth column of this matrix shows the loads which should be shed to balance the load and generation (before execution of *LoadShedding.ComDPL*, all elements of the fourth column are zero).

To update the third column of *LoadPriorityList*, as shown in lines 13–15 of Fig. 6.11, *PreDist_LodP.ComDPL* (explained in the sequel) is used to determine the active power of loads just before islanding condition.¹ Also, since zero active power consumption of a load indicates that the load switch is open, the second column of loads which consumes zero power is set to 0 on line 17 of Fig. 6.11.

Figure 6.13 shows the *PreDist_LodP.ComDPL* code whose inputs are *Result*, *str1*, *Tstop*, and *dT*. As mentioned before, the goal of this script is to detect the active power consumed by load *str1* just before islanding condition occurrence. Since load

¹Power consumption of loads is usually voltage and frequency dependant and varies in transient period of post-disturbance in islanding condition. Thus, the consumed power of loads before islanding occurrence is used to select sufficient loads to shed.

```

!variable declaration                                !1
int Nr,i,EndCnd,cmp,cnt,iCount,iCub,opn;          !2
set sLod,sCntnt;                                    !3
string str1,str2;                                  !4
objectOlod,OCub,OSwitch;                           !5
double P;                                         !6
!Create a set of all loads and sort them to their names (A to Z) !7
sLod    = AllRelevant('* ElmLod');    sLod.SortToName(0);      !8
Nr     = LoadPriorityList.NRow();      !number of rows in the 'LoadPriorityList' !9
!"for" loop to find active power consumed by loads and update 'LoadPriorityList' !10
for (i=1;i<=Nr;i=i+1){                         !11
    str1=LoadPriorityList.RowLbl(i); !label of the ith row of 'LoadPriorityList' !12
    PreDist_LodP.Execute(Result,str1,Tstop,dT);   !13
    P=PreDist_LodP:P;
    LoadPriorityList.Set(i,3,P);
    if(P=0){!P=0: load switch is open           !14
        LoadPriorityList.Set(i,2,0);               !15
    }else{                                         !16
        LoadPriorityList.Set(i,2,1);               !17
    }
}
}                                                 !18
}                                                 !19
}                                                 !20
}                                                 !21

```

Fig. 6.11 DPL script of the *RefreshLoadPriorityList.ComDPL*

Matrix	Name	LoadpriorityList				OK	Cancel
Labels	Matrix:	Priority	In Service?	P	Should Shed?		
	Alachlore	2.	1.	15.	0.	<input checked="" type="checkbox"/>	
	Alpha-Olefins	2.	1.	13.	0.	<input type="checkbox"/>	
	Ammonia	0.	1.	13.	0.	<input type="checkbox"/>	
	Aromatics	1.	1.	12.	0.	<input type="checkbox"/>	
	Butadiene	2.	1.	2.	0.	<input type="checkbox"/>	
	Butanol	1.	1.	50.	0.	<input type="checkbox"/>	
	Benzene	1.	1.	20.	0.	<input type="checkbox"/>	
	Acid Acetic	2.	1.	5.	0.	<input type="checkbox"/>	
	Butenes	1.	1.	15.	0.	<input type="checkbox"/>	
	Carbon Monoxide	1.	1.	2.	0.	<input type="checkbox"/>	
	Ethylene	2.	1.	12.	0.	<input type="checkbox"/>	
	Ethylene Oxide	1.	1.	30.	0.	<input type="checkbox"/>	
	Future I	0.	1.	2.	0.	<input type="checkbox"/>	
	Future II	0.	1.	40.	0.	<input type="checkbox"/>	

Fig. 6.12 Structure of the *LoadpriorityList* matrix

shedding procedure is executed after receiving the notification of islanding occurrence (*islanding* = 1 in lines 30 and 31 of Fig. 6.7), the values measured at $t = T_{stop}-dT$ and $t = T_{stop}$ are related to pre-disturbance and post-disturbance operating points, respectively. Thus, to achieve the pre-disturbance active power of loads, the sample corresponding to $t = T_{stop} - dT$ should be determined. To this purpose, on lines 13–16 of Fig. 6.13, a *do-while* loop is used to find the *n*th sample related to last measured data before $t = T_{stop} - dT$. Also, on lines 18–20, curve number related to the active power consumed (*m:P:bus1*) by load *str1* is identified. Finally, the last measured

```

!variable declaration                                !1
int Nval,n,ivar,cmp;                            !2
double t,Tevnt;                                !3
object Olod;                                    !4
set sLod;                                       !5
string str2;                                    !6
!Create a set of all loads and sort them to their names (A to Z)    !7
sLod = AllRelevant('*._ElmLod');    sLod.SortToName(0);          !8
!Find n: it corresponds to last time that the data are saved before islanding   !9
Tevnt=Tstop-dT;
Nval = ResNval(Result,0);      !Number of values stored in Results !10
n=Nval;
do{
    n=n-1;                                         !11
    GetResData(t, Result, n);!Get the time of nth sample           !12
}while (t>Tevnt)                                !13
!Find ivar:ivar is curve number related to active power consumption of load str1 !14
str2 = sprintf('%s%',str1,'_ElmLod');!str2=str1.ElmLod           !15
Olod = sLod.FirstFilt(str1);        !Olod is an obeject that points to load str1 !16
ivar = ResIndex(Result, Olod, 'm:P:bus1');       !17
!Find the result data corresponding to row n and column ivar in Result      !18
GetResData(P , Result, n,ivar);!Get the (n,ivar) sample from Results      !19
!20
!21
!22

```

Fig. 6.13 DPL script of the *PreDist_LodP.ComDPL*

active power consumed by load *str1* before $t = T_{stop} - dT$ is obtained using *GetResData* function on line 22.

SelectLoadsToShed.ComDPL:

The objective of this script is to select a set of loads to be shed to balance load and generation. As shown in Fig. 6.9, after execution of *IdentifyGenGeneration.ComDPL* and *RefreshLoadPriorityList.ComDPL*, which are described above, *SelectLoadsToShed.ComDPL* is executed on line 13. Inputs of this script are:

- *LoadPriorityList*: it points to *LoadpriorityList* matrix which has been updated in *RefreshLoadPriorityList.ComDPL*.
- *Ptransfer*: it shows the active power passes through the transformer *GridTrns1* (which connects the plant to the grid) before islanding condition.
- *NG*: it is the output of *IdentifyGenGeneration.ComDPL* and indicates the number of generators which are in running mode and produce power.
- *SPRes*: this parameter is the output of *IdentifyGenGeneration.ComDPL* and shows the active power reserve of generators which are in running mode.

To prevent instability after islanding, according to Eq. (1), a load set which totally consumes *Pthr* ($=P_{transfer} - NG \times SPRes$) MW should be shed to balance load and generation. In this regard, a *do-while* loop is used in lines 8–25 of Fig. 6.14 to select a load set. In this loop, starting from the first (top) row of *LoadPriorityList* matrix, loads with priority 1 are added to selection until the sum of the active power consumption of selected loads, *sumP*, becomes greater than *Pthr* (on line 13 of Fig. 6.14, the fourth column of selected loads is set to 1). If the total active power of all loads with priority 1 becomes smaller than *Pthr*, then loads with next priority are added to the selection.

```

!variable declaration                                !1
int Nr,iPriority,Priority,Inservice,EndCnd,i;      !2
string str;                                         !3
double sumP,Pthr;                                    !4
Nr = LoadPriorityList.NRow(); !number of row of LoadPriorityList matrix    !5
Pthr=Ptransfer-SPres*NG; !Pthr MW should be shed to balance load and generation !6
i=1;iPriority=1;EndCnd=0;sumP=0;                      !7
do{                                                 !8
    Priority=LoadPriorityList.Get(i,1);!Get the priority of ithload          !9
    Inservice = LoadPriorityList.Get(i,2);!Inservice=1:load is in sevice       !10
    if ({Priority==iPriority}.and.{Inservice==1}){                         !11
        P=LoadPriorityList.Get(i,3);!P:active power of selected(high priority)load !12
        LoadPriorityList.Set(i,4,1);!set the 4th column of selected load to 1   !13
        sumP=sumP+P;           !sumP:total active power of selected loads   !14
    }
    if (sumP>=(Pthr)){                                         !15
        EndCnd=1;     !if sumP>=Pthr, the selected loads are sufficient !16
    }elseif ({sumP<Pthr}.and.{i<=Nr}){
        i=i+1;!increase i to add more loads to selection until sumP>=Pthr !17
    }
    if({sumP<Pthr}.and.{i>Nr}){
        iPriority=iPriority+1;!increase iPriority to select loads with next priority !18
        i=1;
    }
}while(EndCnd==0)                                     !24
                                                               !25

```

Fig. 6.14 DPL script of the *SelectLoadsToShed.ComDPL*

At the end of this script, in *LoadPriorityList* matrix, the fourth column of selected loads (which totally consume more than *Pthr* MW) are set to one. These selected loads will be shed in *OpenLodSwitch.ComDPL* to balance load and generation.

OpenLodSwitch.ComDPL:

The objective of this script is to shed the set of loads which is selected in *SelectLoadsToShed.ComDPL* script. As shown in line 15 of Fig. 6.9, *LoadPriorityListmatrix* and *Tstop* are applied to *OpenLodSwitch.ComDPL* to open switches of selected loads (in *LoadPriorityList* matrix, the fourth column of these loads are set to 1). In *OpenLodSwitch.ComDPL*, a *for* loop is used to find loads selected in *SelectGensToShed.ComDPL* (line 14 in Fig. 6.15) and then switch events are created to shed the selected loads (line 19-23). Also in *LoadPriorityList* matrix, the second and fourth columns of these loads are set to 0 (lines 24 and 25) which means that they are no longer in service.

6.3.3.3 GenerationShedding.ComDPL

After islanding occurrence (*Islanding* = 1), provided that the generation is more than load demand in the islanded plant (i.e. *Ptransfer* < 0), *GenerationShedding.ComDPL* is executed to select and shed some generators (based on the priority list) to balance generation and load. Then, this script changes operating mode of other generators to LOSC mode to rapidly decrease frequency deviation. Based on lines

```

!variable declaration                                !1
set Slod,SCntnt;                                !2
object Inc,EventDir,event,Olod;                  !3
int Nr,i,ShouldBeShed;                           !4
string str1,str2;                               !5
Inc      = GetCaseCommand('ComInc');            !6
EventDir = Inc:p_event;                         !7
!Create a set of all loads and sort them to their names (A to Z) !8
Slod   = AllRelevant('.ElmLod');    Slod.SortToName(0);        !9
Nr = LoadPriorityList.NRow();                   !10
!Find and shed those Loads identified by LoadPriorityList(i,4)=1; !11
for (i=1;i<=Nr;i=i+1){                         !12
    ShouldBeShed = LoadPriorityList.Get(i,4);     !13
    if(ShouldBeShed=1){!those loads should be shed is identified by 'ShouldBeShed=1'!14
        str1=LoadPriorityList.RowLbl(i);!str1 is the load name which should be shed !15
        str2=sprintf('%s%s',str1,'.ElmLod');          !16
        Olod=Slod.FirstFilt(str2);    !Olod has been selected to be shed           !17
        !create Switch Event to shed the selected loads at t= Tstop           !18
        event      = EventDir.CreateObject('EvtSwitch','Switch Event');       !19
        event:p_target = Olod;                                         !20
        event:mtime   = 0;                                           !21
        event:time    = Tstop;                                         !22
        event.Execute();                                         !23
        LoadPriorityList.Set(i,2,0); !2nd column of selected loads is set to 0 !24
        LoadPriorityList.Set(i,4,0); !4th column of selected loads is set to 0 !25
    }
}                                                 !26
}                                                 !27

```

Fig. 6.15 DPL script of the *OpenLodSwitch.ComDPL*

31 and 34 of Fig. 6.7, inputs of *GenerationShedding.ComDPL* are similar to inputs of *LoadShedding.ComDPL* mentioned in Sect. 6.3.3.2.

As shown in Fig. 6.3, *GenerationShedding.ComDPL* includes four subscripts and a generator priority list (*GenpriorityList.IntMat*). In this script, *RefreshGenPriorityList.ComDPL* is the first subscript executed on line 5 of Fig. 6.16 to update *GenPriorityList* matrix. Then, *SelectGensToShed.ComDPL* is executed on line 7 to select a list of generators (according to the priority list) which should be shed to prevent over frequency. This list is used by *OpenGenSwitch.ComDPL* (line 9 of Fig. 6.16) to shed selected generators. Finally, *ChangeGenMode.ComDPL* is executed on line 11 to change governor mode of generators which are in service from LOLC mode to LOSC mode to rapidly decrease frequency deviation.

```

!variable declaration                                !1
object GenPriorityList;                          !2
GenPriorityList = GenpriorityList;                !3
!Refresh priority list of generators             !4
RefreshGenPriorityList.Execute(Result,GenPriorityList,Tstop,dT); !5
!Select a generator set to shed according to Priority List !6
SelectGensToShed.Execute(GenPriorityList,Ptransfer); !7
!shed selected generators                        !8
OpenGenSwitch.Execute(GenPriorityList,Tstop);      !9
!Change governor mode of other generators from LOLC to LOSC mode !10
ChangeGenMode.Execute(Tstop);                    !11
GenShedNeeded=0;                                !12

```

Fig. 6.16 DPL script of the *GenerationShedding.ComDPL*

RefreshGenPriorityList.ComDPL:

Figure 6.17 shows this script which is used to update the third column of *GenPriorityList*. To update the third column of *GenPriorityList*, as shown in lines 9–11 of Fig. 6.17, *PreDist_GenP.ComDPL* (explained in the sequel) is used to determine the active power of generators just before islanding condition.

The first column of *GenPriorityList*, whose structure is shown in Fig. 6.18, identifies the priority of generators. According to this figure, generator G2 whose priority is 1 is the first machine which will be selected to shed. The second column of this matrix indicates whether a generator is in service or not. Also, the third column shows the active power of each generator just before islanding condition. Moreover, the fourth column shows generators which should be shed to balance load and generation (before execution of *GenerationShedding.ComDPL*, all elements of the fourth column are zero).

Figure 6.19 shows the *PreDist_GenP.ComDPL* code whose inputs are *Result*, *str1*, *Tstop*, and *dT* (line 9 of Fig. 6.17). The goal of the script is to determine the active power of generator *str1* just before islanding condition occurrence. As explained in Sect. 6.3.3.2, to obtain the power of generator *str1* just before islanding

```

!variable declaration
int Nr,i;
string str1;
double P;
Nr = GenPriorityList.NRow();
!"for" loop to find active power of generators and update 'GenPriorityList'
for (i=1;i<=Nr;i=i+1){
    str1=GenPriorityList.RowLbl(i); !label of the ith row of 'GenPriorityList'
    PreDist_GenP.Execute(Result,str1,Tstop,dT);
    P=PreDist_GenP:P;
    GenPriorityList.Set(i,3,P);
    if(P==0){
        GenPriorityList.Set(i,2,0);      !P=0: means load switch is open
    }else{
        GenPriorityList.Set(i,2,1);
    }
}

```

Fig. 6.17 DPL script of the *RefreshGenPriorityList.ComDPL*

	Priority	In Sevice?	P	Should Shed?
G1	2.	1.	120.	0.
G2	1.	1.	120.	0.
G3	3.	1.	120.	0.
G4	4.	1.	120.	0.
G5	5.	1.	120.	0.

Fig. 6.18 Structure of *GenpriorityList* matrix

```

!variable declaration                                !1
int Nval,n,ivar;                                 !2
double t,Tevnt;                                  !3
object OSym;                                    !4
set sSym;                                       !5
string str2;                                    !6
!Create a set of all generators and sort them to their names (A to Z) !7
sSym = AllRelevant('.ElmSym');      sSym.SortToName(0);           !8
!Find n: it corresponds to last time that the data are saved before islanding !9
Tevnt=Tstop-dT;
Nval = ResNval(Result,0);      !number of values stored in result !10
n=Nval;
do{
    n=n-1;                                         !11
    GetResData(t, Result, n);                      !12
}while (t>Tevnt)                                !13
!Find ivar:ivar is curve number related to active power of generator str1 !14
str2 = sprintf("%s%",str1,'.ElmSym');           !15
OSym = sSym.FirstFilt(str1);                     !16
ivar = ResIndex(Result, OSym, 'm:P:bus1');       !17
!Find the result data corresponding to row n and column ivar in result !18
GetResData(P , Result, n,ivar);                  !19
                                                !20
                                                !21
                                                !22

```

Fig. 6.19 DPL script of the *PreDist_GenP.ComDPL*

occurrence, the sample corresponding to $t = T_{stop} - dT$ should be identified by *PreDist_GenP.ComDPL*. The commands of *PreDist_GenP.ComDPL* are similar to *PreDist_LodP.ComDPL* which is shown in Fig. 6.13.

SelectGensToShed.ComDPL:

According to Fig. 6.16, after execution of *RefreshGenPriorityList.ComDPL*, which is described above, *SelectGensToShed.ComDPL* is executed on line 7 to select a set of generators. The selected generators then will be shed to balance load and generation in the islanded plant. Inputs of this script are:

- *GenPriorityList*: it points to *GenpriorityList* matrix which has been updated in *RefreshGenPriorityList.ComDPL*.
- *Ptransfer*: it shows the active power passes through transformer *GridTrns1* (which connects the plant to the grid) before islanding condition. Thus, it indicates the amount of extra power produced by generators in islanded plant.

To prevent over frequency, PMS selects and sheds a set of generators which is totally producing *Ptransfer* (MW). Nevertheless, if LOSC mode exists in governor logic, the alternative approach is to shed fewer generators and change operating mode of other ones to LOSC mode. In this regard, in *SelectGensToShed.ComDPL*, a set of generators which is totally producing 25% of *Ptransfer* is selected to shed (operating mode of other generators will be changed from LOLC mode to LOSC mode in *ChangeGenMode.ComDPL* explained in the sequel).

As shown in Fig. 6.20, in a *do-while* loop (lines 8–25), a set of generators, starting from priority 1, which totally produces $P_{thr}(=0.25 \times P_{transfer})$ MW is selected. In *GenPriorityList* matrix, the fourth column of selected generators is set to 1 on line 13, which will be used by *OpenGenSwitch.ComDPL* to shed selected generators.

```

!variable declaration
int Nr,iPriority,Priority,Inservice,EndCnd,i;
string str;
double sumP,Pthr;
Nr = GenPriorityList.NRow();
Pthr=0.25*abs(Ptransfer); !Pthr MW should be shed to balance load and generation
i=1;iPriority=1;EndCnd=0;sumP=0;
do{
    Priority = GenPriorityList.Get(i,1);!Get the priority of ith gen.in matrix
    Inservice = GenPriorityList.Get(i,2);!Inservice=1:gen.is in running mode
    if ({Priority=iPriority}.and.{Inservice=1}){
        P=GenPriorityList.Get(i,3);!P:active power of selected(high priority)gen.
        GenPriorityList.Set(i,4,1);!set the 4th column of selected gen. to 1
        sumP=sumP+P;           !sumP: total active power of selected loads
    }
    if(sumP>=Pthr){
        EndCnd=1;      !if sumP>=Pthr, the selected gen.s are sufficient
    }elseif ({sumP<Pthr}.and.{i<=Nr}){
        i=i+1;!increase i to add more gen.s to selection until sumP>=Pthr
    }
    if({sumP<Pthr}.and.{i>Nr}){
        iPriority=iPriority+1;!increase iPriority to select gen.s with next priority
        i=1;
    }
}while(EndCnd=0)

```

Fig. 6.20 DPL script of the *SelectGensToShed.ComDPL*

OpenGenSwitch.ComDPL:

This script is used to shed those generators which are selected in *SelectGensToShed.ComDPL* (in *GenPriorityList* matrix, the fourth column of these generators are set to 1). As shown in line 9 of Fig. 6.16, inputs of this script are:

- *GenPriorityList*: it points to *GenpriorityList* matrix.
- *Tstop*: it corresponds to the current operating time.

In *OpenGenSwitch.ComDPL*, a *for* loop is used to find generators which are selected in *SelectGensToShed.ComDPL* (line 14 in Fig. 6.21) and then switch events are created to shed the selected generators (line 19–23). Also in *GenPriorityList* matrix, the second and forth columns of these generators are set to 0 (lines 24 and 25) which means that they are no longer in service.

ChangeGenMode.ComDPL:

This script is used to change operation mode of generators which are in running mode (have not been shed in *OpenGenSwitch.ComDPL*) from LOLC to LOSC mode to rapidly change their generation and prevent over frequency. In governor used in the test system, when parameter *change* is 0, generator works in LOLC mode and when it is set to 1, operating mode of generator change to LOSC. Thus, to change operating mode of generators to LOSC, a *for* loop is used (lines 10–22 of Fig. 6.22) to find the governor of generators which are in running mode and create parameter event (*EvtParam*) to set the parameter *change* of governor to 1.

```

!variable declaration                                !1
set sSym;                                         !2
object OSym,event,Inc,EventDir;                   !3
int Nr,i,ShouldBeShed;                           !4
string str1,str2;                               !5
Inc      = GetCaseCommand('ComInc');             !6
EventDir = Inc:p_event;                         !7
!Create a set of all generators and sort them to their names (A to Z) !8
sSym = AllRelevant('*._ElmSym');    sSym.SortToName(0);           !9
Nr   = GenPriorityList.NRow();                  !10
!Find and shed those generators identified by GenPriorityList(i,4)=1; !11
for (i=1;i<=Nr;i=i+1){                         !12
    ShouldBeShed = GenPriorityList.Get(i,4);
    if(ShouldBeShed=1){!generators should be shed is identified by 'ShouldBeShed=1' !13
        str1=GenPriorityList.RowLbl(i);
        str2=sprintf('%s%s',str1,'._ElmSym');
        OSym=sSym.FirstFilt(str2); !Osym points to the gen. which should be shed !17
        !create Switch Event to shed the selected generators at t= Tstop !18
        event      = EventDir.CreateObject('EvtSwitch','Switch Event');       !19
        event:p_target = OSym;
        event:mtime   = 0;
        event:time    = Tstop;
        event.Execute();
        GenPriorityList.Set(i,2,0); !2nd column of selected generators is set to 0 !24
        GenPriorityList.Set(i,4,0); !4rd column of selected generators is set to 0 !25
    }
}                                                 !26
                                                !27

```

Fig. 6.21 DPL script of the *OpenGenSwitch.ComDPL*

```

!variable declaration                                !1
object O,event,EventDir,Inc;                     !2
sets Dsl;                                         !3
string str;                                       !4
int i;                                            !5
sDsl     = AllRelevant('*._ElmDsl');    sDsl.SortToName(0);           !6
Inc      = GetCaseCommand('ComInc');             !7
EventDir = Inc:p_event;                         !8
!Create Parameter Event to change load control mode to speed control !9
for (O=sDsl.First();O;O=sDsl.Next()){          !10
    str = O:loc_name;
    i = strstr(str, '_pcu');
    if({i>=0}.and.{O:outserv=0}){
        event = EventDir.CreateObject('EvtParam','Change Mode Event'); !14
        event:variable = 'change';
        event:p_target = O;
        event:value   = '1';
        event:mtime   = 0;
        event:time    = Tstop;
        event.Execute();
    }
}
                                                !21
                                                !22

```

Fig. 6.22 DPL script of the *ChangeGenMode.ComDPL*

6.3.3.4 SteadyStateDetection.ComDPL

As shown in lines 30–36 of Fig. 6.7, after islanding detection (*Islanding* = 1), either load shedding or generation shedding is executed to prevent instability which causes operating point to reach to a stable equilibrium point. Nevertheless, in this

equilibrium point, some system parameters, such as frequency, may be out of acceptable range which means that remedial actions should be carried out to improve operating point. Since these remedial actions are performed in steady-state condition, *SteadyStateDetection.ComDPL* is executed on line 38 of Fig. 6.7 to detect steady-state condition. As shown in this figure, the inputs to this script are:

- *Result*: points to *Results.ElmRes* where the results of the simulation are stored.
- *Tstop*: it corresponds to the current operating time.

According to Fig. 6.23, on lines 11–17 of *SteadyStateDetection.ComDPL*, curve number related to the frequency of the bus connected to the first load (*m:fe*) is identified. Then, on lines 19–30, the average system frequency during last three seconds (*favr*) is calculated. Finally, as shown in lines 32–40, if the deviation of system frequency from its average value, *favr*, be less than a pre-specified value (here 0.0005 p.u.), *SteadyState* parameter is set to 1 which means that system is in steady-state

```

! variable declaration
object O,Obus,Oload;
string str;
set sLod,allBars;
int ivar,Nval,n,i;
double t,f,T,fsum;
sLod = AllRelevant('.ElmLod');           sLod.SortToName(0);          !1
allBars = AllRelevant('.StaBar,*.ElmTerm'); allBars.SortToName(0);        !2
Nval = ResNval(Result,0);    !number of values stored in result      !3
!Find ivar:ivar is curve number related to freq. of bus connected to first load !4
Oload=sLod.First();!select the first load                                !5
Oload=Oload.First();!select the first load                                !6
Obus=O:cBusBar;      !select the bus connected to the first load       !7
str=Obus:loc_name;!str: name of the bus connected to the first load     !8
str = sprintf('%s%',str,'.ElmTerm');                                     !9
Obus = allBars.FirstFilt(str);!Obus: points to bus connected to first load !10
ivar = ResIndex(Result, Obus, 'm:fe');!curve number related to the freq. of Obus !11
!Calculate average frequency ('favr') of system during the last 3s:          !12
T=Tstop-3;                                         !13
fsum=0;                                           !14
n=Nval;                                         !15
i=0;                                              !16
do{                                                 !17
    n=n-1;                                         !18
    i=i+1;                                         !19
    GetResData(t, Result, n);                      !20
    GetResData(f , Result, n,ivar);                !21
    fsum=fsum+f;                                    !22
}while (t>T)                                       !23
favr=fsum/i;                                       !24
!Calculate maximum frequency deviation from 'favr' during the last 3s:      !25
SteadyState=1;                                     !26
n=Nval;                                         !27
do{                                                 !28
    n=n-1;                                         !29
    GetResData(f , Result, n,ivar);                !30
    if({f>(favr+0.0005)}.or.{f<(favr-0.0005)}){ !31
        SteadyState=0;                            !32
    }
}while(n>(Nval-i))                                !33
!40

```

Fig. 6.23 DPL script of the *SteadyStateDetection.ComDPL*

condition. Finally, *SteadyState* and *favr* are sent to *TimeDomainSimulation.ComDPL* as output (lines 39 and 40 of Fig. 6.7).

6.3.3.5 RegulateFrequency.ComDPL

When the system reaches to steady-state operating point, *RegulateFrequency.ComDPL* is executed on line 43 of Fig. 6.7 to perform secondary frequency control and precisely regulate frequency. Inputs of this script are:

- *favr*: it is the average system frequency during the last three seconds.
- *Tstop*: it corresponds to the current operating time.
- *TRF*: it is the last time that *RegulateFrequency.ComDPL* has been executed.

As shown in Fig. 6.24, on line 17, the current setting of governors which are in running mode is identified (It worth noting that in this test system, the setting of the governor is obtained using *wref2*). To regulate frequency, if *favr* is more (less) than nominal value, the governor setting should be decreased (increased) by *dw* ($=0.005$ p.u.) to regulate frequency. Thus, new setting of the governor is calculated using equations mentioned on lines 19 or 21. Finally, on lines 23–31, parameter event (*.EvtParam) is created to change governor setting and *TRF* is set to *Tstop*. It should be stated that a proper value for *dw* can be obtained using offline simulations, and it should be small enough to prevent the oscillatory behaviour.

6.4 Simulation Results

Here, two scenarios are considered to show how those scripts mentioned previously, which simulate PMS logic, can improve operating point in the test system. It should be stated that before execution of these scripts, in *calculation of initial conditions* page, *Results.ElmRes* should be assigned to *Results Variables* (see Fig. 6.25).

6.4.1 Scenario 1 (3GT + Grid Connected + Import Power)

In this scenario, three generators produce 276 MW, the plant imports about 117 MW from the grid, and load demand is 387 MW. At $t = 69.5$ s, the plant disconnects from the grid and PMS logic, which assess islanding occurrence every second, detects islanding condition at $t = 70$ s. Since, before islanding condition, the plant imports power from the grid ($P_{transfer} > 0$), load shedding procedure is executed (see lines 30–31 in Fig. 6.7) to shed the required amount of load calculated in *SelectLoadsToShed.ComDPL* script (Fig. 6.14) according to Eq. (1).

```

! variable declaration                                !1
string str,sw;                                     !2
object O,event,Inc,EventDir;                      !3
sets Dsl;                                         !4
int i;                                            !5
double dw,w,wnew;                                 !6
! Define sets and parameters                       !7
sDsl      = AllRelevant('*ElmDsl');      sDsl.SortToName(0);    !8
Inc       = GetCaseCommand('ComInc');           !9
EventDir = Inc:p_event;                         !10
dw=0.005;                                         !11
!Create Parameter Event (EvtParam) to regulate frequency !12
for (0=sDsl.First();0<sDsl.Next()){             !13
    str = O:loc_name;                           !14
    i = strstr(str, 'pcu');                     !15
    if({i>0}.and.{0:outserv==0}){
        w=0:s:wref2;                          !16
        if(favr>1){                         !17
            wnew=w-dw;                        !18
        }else{                               !19
            wnew=w+dw;                        !20
        }
        sw=sprintf('%f',wnew);                !21
        event   = EventDir.CreateObject('EvtParam', 'Change Frequency'); !22
        event:variab = 'wref2';              !23
        event:p_target = 0;                  !24
        event:p_value = sw;                 !25
        event:mtime = 0;                   !26
        event:time = Tstop;                !27
        event.Execute();                  !28
        TRF=Tstop;                         !29
    }
}
}                                                     !30
}                                                     !31
}                                                     !32
}                                                     !33

```

Fig. 6.24 DPL script of the *RegulateFrequency.ComDPL*

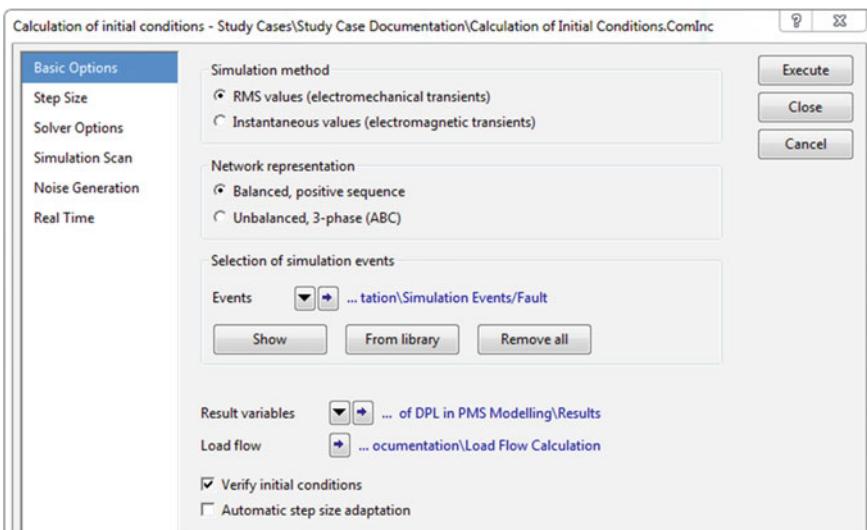


Fig. 6.25 *Results.ElmRes* should be assigned to *Results Variables*

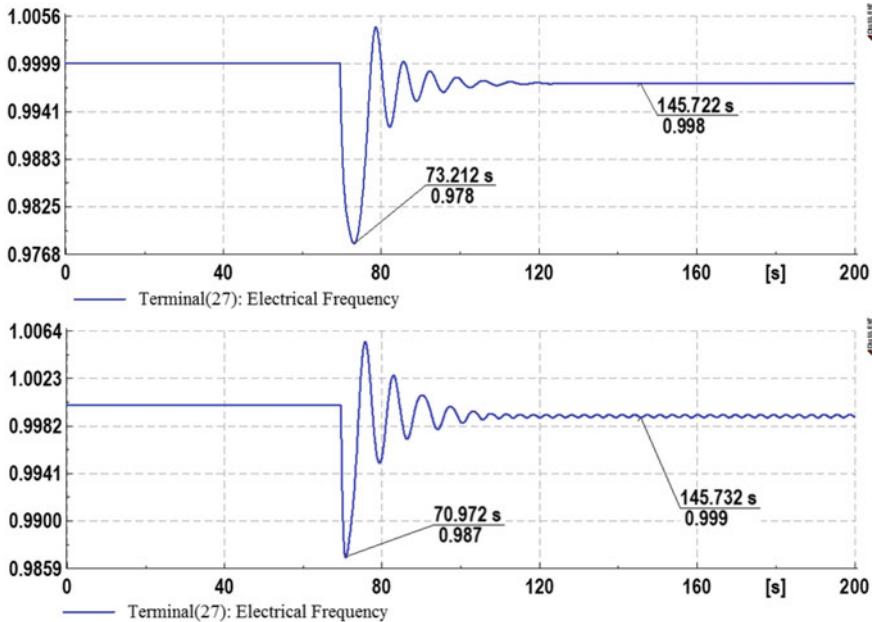


Fig. 6.26 Load shedding assuming $SPRes = 7$ (Top) and $SPRes = 0$ (bottom)

This procedure causes four loads (which totally consume 97 MW) to be shed. The top plot in Fig. 6.26 shows that although the post-disturbance frequency is less than nominal frequency, the load shedding procedure can properly mitigate under frequency condition and cause the frequency to reach to 49.9 Hz in steady-state condition. It should be mentioned that neglecting $SPRes$ in load shedding procedure (i.e. $SPRes = 0$ in line 6 of Fig. 6.14) shows more effective results and causes the frequency to reach to 49.96 Hz in post-disturbance steady-state condition (bottom plot in Fig. 6.26).

In addition to the amount of load which should be shed, the reaction time of PMS is important. When the PMS checks the islanding condition every 5 s, which means that the reaction time of PMS increases to 4.5 s, Fig. 6.27 shows that the minimum frequency reaches to 45.45 Hz which may result in operation of frequency relays and more economic losses.

6.4.2 Scenario 2 (5GT + Grid Connected + Export Power)

In this case, five generators which totally produce 600 MW are in running mode, the plant exports about 398 MW to the external grid, and the total active power of loads which are in service is 198 MW. At $t = 69.5$ s, the plant disconnects from the grid which is detected at $t = 70$ s by *IslandingDetection.ComDPL*. In this

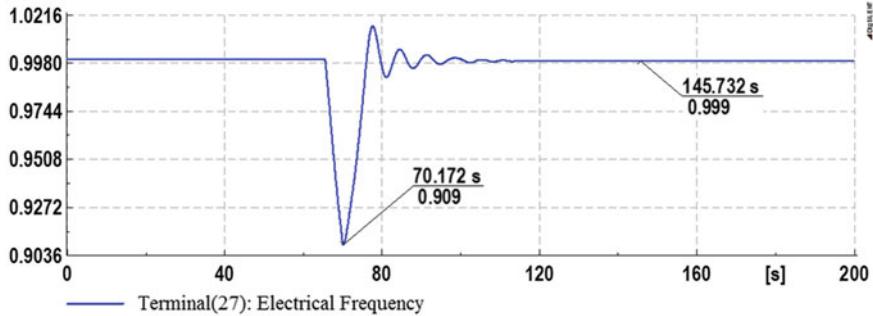


Fig. 6.27 Increasing the PMS reaction time results in the lower frequency

Table 6.1 Priority list of generators used in Scenario 2

Generator	G1	G2	G3	G4	G5
Priority	2	1	3	5	4

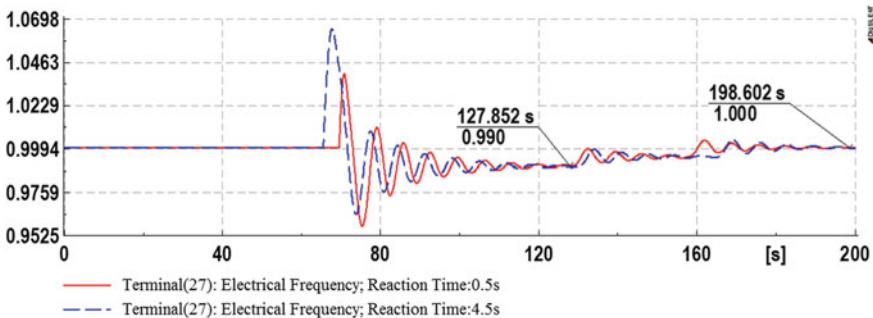


Fig. 6.28 *GenerationShedding.ComDPL* prevents over frequency and *RegulateFrequency.ComDPL* regulates frequency after islanding occurrence

condition, since $P_{transfer} < 0$ before islanding occurrence, generation shedding procedure is executed on line 34 of Fig. 6.7 to shed generator G2 (according to Table 6.1). Also, the generation mode of other generators are changed from LOLC to LOSC mode which causes system frequency to become about 49.5 Hz at around $t = 128$ s (Fig. 6.28). Then, in steady-state condition, *RegulateFrequency.ComDPL* is executed at about $t = 128$ and $t = 158$ s to regulate frequency. Finally, system frequency reaches to 50 Hz at about $t = 200$ s (see Fig. 6.28). As shown in this figure, when the reaction time increase from 0.5 to 4.5 s, the frequency deviation increases which may cause over frequency relays to operate.

6.5 Conclusion

PMS is an intelligent system which assesses abnormal conditions and contingency occurrence and assists system operators in executing timely proper remedial actions to regain operating point to the acceptable region. Since PMS executes automatic control actions to improve the operating point, performing accurate offline simulations to assess the effectiveness of considered PMS logic is inevitable. In this regard, the main aim of this study is to investigate the ability of DPL in modelling the PMS logic. Considering the dynamic model of power system equipment, simulation results in an industrial test system show that power factory can properly simulate PMS logic. According to these results, just after contingency occurrence, PMS logic detects islanding condition occurrence and executes load shedding or generation shedding to prevent instability. Also, after reaching to steady-state condition, the considered PMS logic executes secondary frequency control procedure to precisely regulate the system frequency.

References

1. K.G. Ravikumar, T. Alghamdi, J. Bugshan, S. Cott Manson, S.K. Raghupathula, Complete power management system for an industrial refinery. *IEEE Trans. Ind. Appl.* **52**(4), 3565–3573 (2016)
2. B. Belvedere, M. Bianchi, A. Borghetti, C.A. Nucci, M. Paolone, A. Peretto, A microcontroller-based power management system for standalone microgrids with hybrid power supply. *IEEE Trans. Sustain. Energy* **3**(3), 422–431 (2012)
3. A. Parizad, *Dynamic Stability Analysis for Damavand Power Plant Considering PMS Functions by DIGSILENT Software*. Environment and Electrical Engineering (EEEIC), 13th International Conference (2013)
4. K. Gray, J. Kumm, J. Mraz, *A High-Level Framework for Implementation and Test of IEC 61850-Based Microgrid Power Management Systems*. IEEE/PES Transmission and Distribution Conference and Exposition (2016)
5. F. Pacheco, H. Refaat, D. Pescosolido, *Power Management System in an Industrial Plant*. Petroleum and Chemical Industry Technical Conference (PCIC) (2012)
6. S. St. Iliescu, I. Făgărăşan, V. Popescu, C. Soare, Gas turbine modeling for load frequency control. *U.P.B. Sci. Bull. Series C* **70**(4) (2008)

Chapter 7

Determining Wide-Area Signals and Locations of Regulating Devices to Damp Inter-Area Oscillations Through Eigenvalue Sensitivity Analysis Using DIgSILENT Programming Language



Horacio Silva-Saravia, Yajun Wang and Héctor Pulgar-Painemal

Abstract This chapter introduces the concept of eigenvalue sensitivity to analyse the installation location and feedback signals of damping regulating devices using DIgSILENT Programming Language. A state-space representation of the linearized system is estimated by dynamic simulations and used to provide two indices based on mode controllability and mode observability. By inspection of these indices, the number of location/signal candidates is reduced, decreasing the computational cost of modal analysis and time-domain simulation for the final selection of regulating device location and feedback signals. This approach is applied to determine the location/signal of a Battery Energy Storage (BES) regulating the inter-area oscillation between the Northern Chile Interconnected System (NCIS) and the Argentinian Interconnected System (AIS).

Keywords Controllability · Eigenvalue sensitivity · Electromechanical oscillations · Energy storage systems · FACT devices · Observability · State-space model

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_7) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

H. Silva-Saravia · Y. Wang · H. Pulgar-Painemal (✉)

Electrical Engineering & Computer Science, The University of Tennessee, Knoxville,
Knoxville, TN 37996-2250, USA

e-mail: hpulgar@utk.edu

H. Silva-Saravia

e-mail: hsilvasa@vols.utk.edu

Y. Wang

e-mail: ywang139@vols.utk.edu

7.1 Introduction

Smart grids take advantage of communication technologies and computer-based remote control and automatization over a wide range of power system devices such as breakers, capacitors, transformers and even Flexible Alternate Current Transmission Systems (FACTS) and Energy Storage Systems (ESS). In particular, FACTS and ESS may play an important role as damping regulating devices in the control of inter-area oscillations, which are expected to appear more frequently in the system as a consequence of a relative system inertia reduction.

These damping regulating devices have the advantage to be installed virtually anywhere in the power system, allowing better performances in comparison with Power System Stabilizers (PSS)—which may work well for local rather than inter-area oscillations. Therefore, installation location can be determined as a design parameter when considering system dynamics [1]. Also, wide-area measurements as feedback signals for the controllers are proven to have a positive impact on the performance [2, 3]. Thus, feedback signals may be considered as a design parameter choosing from available measurements coming from Phasor Measurement Units (PMU) or similar devices. Although a common analysis, exhaustive time-domain simulations over all the possible locations and signal candidates are computationally expensive and unfeasible for real power systems. Hence, an analytical approach is desired.

A screening approach based on eigenvalue sensitivity allows comparing the impact of damping regulating devices for different installation locations and different feedback signals, using a state-space representation of the linearized system under study. The best location/signal candidates are chosen, and for final selection, time-domain simulations using the nonlinear system model are executed. This chapter employs the concept of eigenvalue sensitivity and defines metrics to compare the effects of the location and measurements candidates on the oscillation damping. The approach is coded in DIgSILENT Programming Language and applied to decide the location/signals of a Battery Energy Storage System (BESS) in the Northern Chile Interconnected System (NCIS) when connected to the Argentinian Interconnected System (AIS).

7.2 Eigenvalue Sensitivity

7.2.1 General System Representation

Power systems are large-scale nonlinear systems with complex dynamics and parameter dependencies. In the case of low-frequency oscillations, a recurrent phenomenon in power systems, the dynamics can be fairly represented by a linearized representation around a given equilibrium point. In mathematical form:

$$\Delta\dot{x} = A\Delta x + B\Delta u \quad (7.1)$$

$$\Delta y = C\Delta x + D\Delta u \quad (7.2)$$

where $\Delta x = x - x^e$, $\Delta y = y - y^e$ and $\Delta u = u - u^e$. Here, x , y and u correspond to the vectors of state, output and input variables, respectively—the superscript e indicates that the variables are those at the equilibrium point. A , B , C and D are the system matrices—all with proper dimensions. The zero-input response of this system for each state variable $\Delta x_j(t)$ can be expressed as shown by Pizarro-Gálvez et al. [4] as:

$$\Delta x_j(t) = \sum_{i=1}^n k_{ji} e^{\lambda_i(t-t_0)}, \forall t \geq t_0 \quad (7.3)$$

where λ_i is the i th system eigenvalue and k_{ji} is a corresponding constant. The term $e^{\lambda_i(t-t_0)}$ is the i th system mode. When the eigenvalue is a complex number such as $\lambda_i = \alpha_i + j\beta_i$, the mode corresponds to an oscillation of the form $e^{\alpha_i(t-t_0)} e^{j\beta_i(t-t_0)}$. From here, the concepts of damping, oscillation frequency and structural stability can be defined—check Refs. [4, 5].

7.2.2 Eigenvalue Problem and Parameter Dependency

The eigenvalue problem can be stated as:

$$(A - \lambda_i I)v_i = 0 \quad (7.4)$$

$$\omega_i^T (A - \lambda_i I) = 0 \quad (7.5)$$

where v_i and ω_i are the right and left eigenvectors associated with the i th eigenvalue λ_i . Note that the matrix A is dependent on the system parameters, which are assumed to be constant during dynamic simulation. When these parameters change, the system matrix A is modified, and consequently, the original eigenvalues and eigenvectors are affected. As an example, consider the gain of a feedback loop controller. The change of this parameter can potentially displace a critical eigenvalue of interest; if the displacement is towards the negative direction of the real axis, a desire damping ratio may be achieved, or if the displacement is towards the positive direction, the damping ratio would be reduced in such a way that the eigenvalue may even cross the imaginary axis making the system unstable. Another interesting example is when, out of different potential actuators, the most effective one to displace critical eigenvalues needs to be chosen. In the same line of thoughts, the actuator signal can be selected; out of several system signals, which signal would be the most effective to displace the critical eigenvalue? To answer these questions, the study of eigenvalue sensitivity under system parameters is crucial and will be addressed next.

Consider the eigenvalue problem again shown in (7.4) and differentiate this with respect to a generic parameter K as follows:

$$\frac{\partial}{\partial K} \{(A - \lambda_i I)v_i\} = (A - \lambda_i I) \frac{\partial v_i}{\partial K} + \left(\frac{\partial A}{\partial K} - \frac{\partial \lambda_i}{\partial K} I \right) v_i = 0 \quad (7.6)$$

Pre-multiplying (7.6) by the i th left eigenvector ω_i^T and assuming that the eigenvectors are normalized, i.e. $\omega_i^T v_i = 1$, the following relationship is obtained:

$$\underbrace{\omega_i^T (A - \lambda_i I)}_0 \frac{\partial v_i}{\partial K} + \omega_i^T \frac{\partial A}{\partial K} v_i - \underbrace{\omega_i^T v_i}_1 \frac{\partial \lambda_i}{\partial K} = 0 \quad (7.7)$$

$$\Rightarrow \frac{\partial \lambda_i}{\partial K} = \omega_i^T \frac{\partial A}{\partial K} v_i \quad (7.8)$$

7.2.3 Mode Controllability and Mode Observability

The well-known expression obtained in the previous section is useful to determine the eigenvalue sensitivity; however, it is not practical in a multi-parameter analysis, since the calculation of the derivative of the matrix A with respect to the parameter K is required. Therefore, a simpler relationship with a more affordable term is needed. The following approach gives an equivalent representation of Eq. (7.8) for a more generic feedback control considering only calculations derived from the original system matrices and the feedback transfer function. Finally, this expression will be generalized to introduce two main factors of the eigenvalue sensitivity: mode controllability and mode observability.

Consider a generic system “ a ” with an output feedback controller called system “ b . * Note that while the controller output (y_b) is the system input (u_a), the controller input (u_b) is the system output (y_a). The state-space representation is:

$$\dot{x}_a = A_a x_a + B_a y_b \quad (7.9)$$

$$y_a = C_a x_a \quad (7.10)$$

$$\dot{x}_b = A_b x_b + B_b y_a \quad (7.11)$$

$$y_b = C_b x_b + D_b y_a \quad (7.12)$$

Alternatively, in compact matrix form:

$$\begin{bmatrix} \dot{x}_a \\ \dot{x}_b \end{bmatrix} = \underbrace{\begin{bmatrix} A_a + B_a D_b C_a & B_a C_b \\ B_b C_a & A_b \end{bmatrix}}_A \begin{bmatrix} x_a \\ x_b \end{bmatrix} \quad (7.13)$$

The matrix A is the closed loop system matrix. Note that, in frequency domain, the controller transfer function can be compactly written as:

$$H(s) = C_b M_b(s) B_b + D_b \quad (7.14)$$

where

$$M_b(s) = (sI - A_b)^{-1} \quad (7.15)$$

Moreover, the derivative of $H(s)$ with respect to the controller in K is denoted as $H'(s)$ and has the following expression:

$$\begin{aligned} H'(s) = & C'_b M_b B_b + C_b M'_b B_b + C_b M_b B'_b + D'_b + C'_b M_b B_b \\ & + C_b M_b A'_b M_b B_b + C_b M_b B'_b + D'_b \end{aligned} \quad (7.16)$$

Note that the augmented left and right eigenvector satisfies (7.4) and (7.5), thus,

$$\begin{bmatrix} A_a + B_a D_b C_a & B_a C_b \\ B_b C_a & A_b \end{bmatrix} \begin{bmatrix} v_{i1} \\ v_{i2} \end{bmatrix} = \lambda_i \begin{bmatrix} v_{i1} \\ v_{i2} \end{bmatrix} \quad (7.17)$$

$$[\omega_{i1}^T \omega_{i2}^T] \begin{bmatrix} A_a + B_a D_b C_a & B_a C_b \\ B_b C_a & A_b \end{bmatrix} = \lambda_i [\omega_{i1}^T \omega_{i2}^T] \quad (7.18)$$

If v_{i1} is the right eigenvector of the open loop system, then $v_{i2} = M(\lambda_i) B_b C_a v_i$. Similarly, assuming ω_{i1}^T is the left eigenvector of the open loop system, then $\omega_{i2}^T = \omega_i^T B_a C_b M(\lambda_i)$.

Finally, by Eq. (7.8), the sensitivity of a closed loop system eigenvalue with respect to the controller gain K is given by:

$$\lambda' = \frac{\partial \lambda_i}{\partial K} = \omega_i^T \frac{\partial A}{\partial K} v_i = [\omega_{i1}^T \omega_{i2}^T] \begin{bmatrix} B_a D'_b C_a & B_a C'_b \\ B'_b C_a & A'_b \end{bmatrix} \begin{bmatrix} v_{i1} \\ v_{i2} \end{bmatrix} \quad (7.19)$$

which can be written as:

$$\begin{aligned} \lambda'_i = & \omega_i^T B_a D'_b C_a v_i + \omega_a^T B_a C_b M(\lambda_i) B'_b C_a v_i + \omega_i^T B_a C'_b M(\lambda_i) B_b C_a v_i \\ & + \omega_i^T B_a C_b M(\lambda_i) A'_b M(\lambda_i) B_b C_a v_i \end{aligned} \quad (7.20)$$

This expression resembles Eq. (7.16) and, in fact, can be factorised to get a more compact relationship. Thus, the eigenvalue sensitivity on a feedback gain, as a function of the original system matrices and the transfer function of the feedback loop, is described by Pagola et al. [6]:

$$\lambda'_i = \omega_i^T B_a H(\lambda_i)' C_a v_i \quad (7.21)$$

$$|\lambda'_i| = |\omega_i^T B_a| \cdot |H(\lambda_i)'| \cdot |C_a v_i| \quad (7.22)$$

where the term $|\omega_i^T B_a|$ is called the mode controllability of mode i (*MC*) since relates the effects of the chosen input in the mode displacement. Similarly, the term $|C_a v_i|$ is called the mode observability of mode i (*MO*) and relates the effects of the chosen measurement signal of the feedback control in the mode displacement.

Given a common transfer function, for example, the same Energy Storage System (ESS) controlling inter-area oscillations with exactly the same feedback gain, then the relative difference in the magnitude of the mode displacement is given by the *MC* and *MO*. The direction of the displacement can be adjusted later by adding phase-lead compensator in the feedback loop considering the necessary compensation angle. In order to compare the effect in the mode displacement of two or more input and two or more measurement signals, the controllability index (*CI*) and the observability index (*OI*) are defined [7]. Let $l \in L$ the set of all possible inputs and $\delta \in \Delta$ the set of all possible feedback measurements. Then the controllability index associated with input l (CI_l) and the observability index associated with signal δ (OI_δ) are expressed as:

$$CI_l = \frac{MC_l}{\max_{l \in L} MC_l} \quad (7.23)$$

and

$$OI_\delta = \frac{MO_\delta}{\max_{\delta \in \Delta} MO_\delta} \quad (7.24)$$

7.3 DPL Scripting and Procedure

All the calculations described in the previous section are programmed in DPL and can be classified into two main parts:

- (a) State-space representation: In order to perform eigenvalue sensitivity, a state-space model of the linearized system is needed. Matrix A gives information of the eigenvalues of interest and the respective left and right eigenvectors as shown in Eqs. (7.3) and (7.4). Matrices B and C are used to calculate the eigenvalue sensitivity Eq. (7.22). In order to allow the users to manipulate some data from the modal analysis calculation, DIgSILENT has included a function to export the system matrix A from the state-space model. This matrix A contains information with concrete physical interpretations (modes), and mathematically speaking, it is uniquely defined. However, matrices B and C of the state-space model are related to the input and output variables; thus, they are

unavailable a priori as the user definition/intervention is required. To provide a framework for obtaining the full numerical linear state-space representation in DIgSILENT, this section introduces a procedure to estimate the input and output matrices for any large system with large number of state variables using DPL.

- (b) Eigenvalue sensitivity: CI and OI are calculated according to Eqs. (7.23) and (7.24) for all prospective installation location and feedback signals. The results are exported as vectors in two independent “csv” files for further analysis.

The following programs are coded for the application to the NCIS; however, a guideline to apply them to any system is given. A flow chart of the whole procedure is shown in Fig. 7.1.

The first step is to run the program *Main_script_1* which calculates the modal analysis to the open loop system. The modal analysis calculation object (ComMod*) is set to export the output to MATLAB files for system matrices, eigenvalues, left eigenvectors, right eigenvectors and participation factors in a customised path. Figure 7.2 shows the script of *Main_script_1*.

The execution of *Main_script_1* creates seven “mtl” files and two “txt” files in the chosen path. In particular, the file “VariableToIdx_Amat.txt” will be read several times in the code to index the results to their respective state variables.

In order to allow a correct reading of the “VariableToIdx_Amat.txt.” file, the header row must be manually deleted with a text editor such as NotePad++. Any other additional description row in the body of the file must also be deleted, and then the file should contain the same number of rows as the number of state

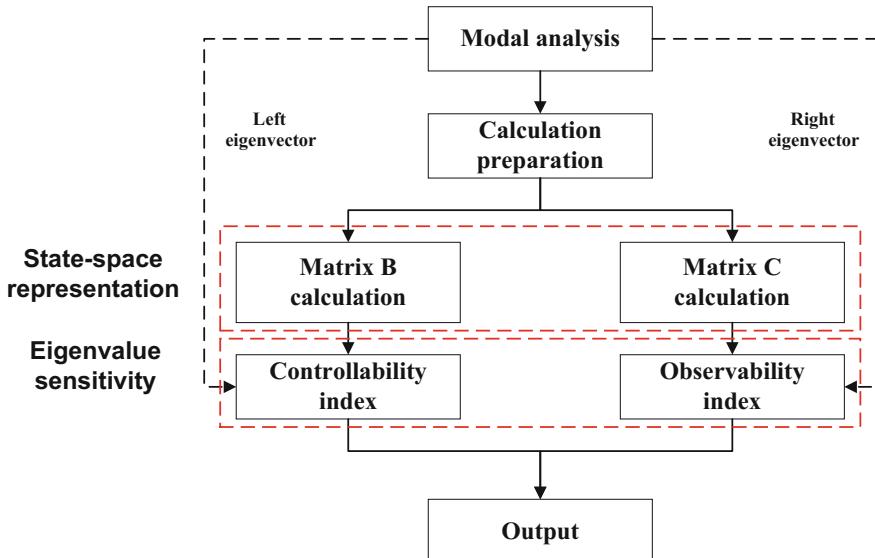


Fig. 7.1 Flow chart for eigenvalue sensitivity calculation

```

## Main_script_1 ##
string path;
set StudyCases;
object Folder,Case,Initial,Modal;

path=' D:\Temp';                                !Set the path of the target folder
Folder = GetProjectFolder('study');              !Get project folder
StudyCases = Folder.GetContents('.IntCase');      !Get all the study cases
StudyCases.SortToName();                         !Sort all the study cases
Case = StudyCases.First();                      !Get the first study case
Case.ShowFullName();                            !Activate the first study case
Case.Activate();
Case.ShowFullName();

Initial=GetCaseCommand ('ComInc');               !Get Calculation of Initial Conditions
Modal=GetCaseCommand ('ComMod');                 !Get object Modal Analysis

!Set the output of the modal analysis
Modal:iLeft=1;
Modal:iRight=1;
Modal:iPart=1;
Modal:iSysMatsMatl=1;
Modal:iEvalMatl=1;
Modal:iREVMatl=1;
Modal:iLEVMatl=1;
Modal:iPartMatl=1;
Modal:dirMatl=path;                            !Set the path of the output files

!Simulation
Initial.Execute();                           !Perform the Calculation of Initial Conditions
Modal.Execute();                             !Perform the Modal Analysis

```

Fig. 7.2 DPL for modal analysis and output path definition

variables. Once this is done, *Main_script_2* can be executed. Here, a preparation program reads the state variables and adds them to the All Calculation (ElmRes*) object in the study case. This allows the next programs to perform the necessary calculations over these variables to estimate the matrices *B* (in *Bcalculations*) and *C* (in *Ccalculations*). After matrices *B* and *C* are calculated and stored as *IntMat** objects in the study case, the programs *Controllability* and *Observability* obtain the mode controllability and mode observability, which are finally used to get and export *CI* and *OI* with the program *Output*. All these five programs are contained inside of the *Main_script_2*.

Figure 7.3 shows a screenshot of the *Main_script_2* element dialog and the contents tab. A general selection (SetSelect*) must be created in the study case and selected in this program. The general selection must contain the prospective installation location buses, which will be used for the calculation of the *CI*.

The DPL script of *Main_script_2* is shown in Fig. 7.4. Note that the path specified must be the same as the one used for *Main_script_1*. This allows the subprograms to read and work with the variables of interest.

As mentioned before, the preparation program reads the file “VariableToIdx_Amat.txt” to add the state variables into “All calculation” and to use them in the *B* and *C* matrix calculation. This step is necessary for large power systems since the number of states may easily exceed a few hundred and manual setting can be cumbersome. The following subprogram uses the typical structure of

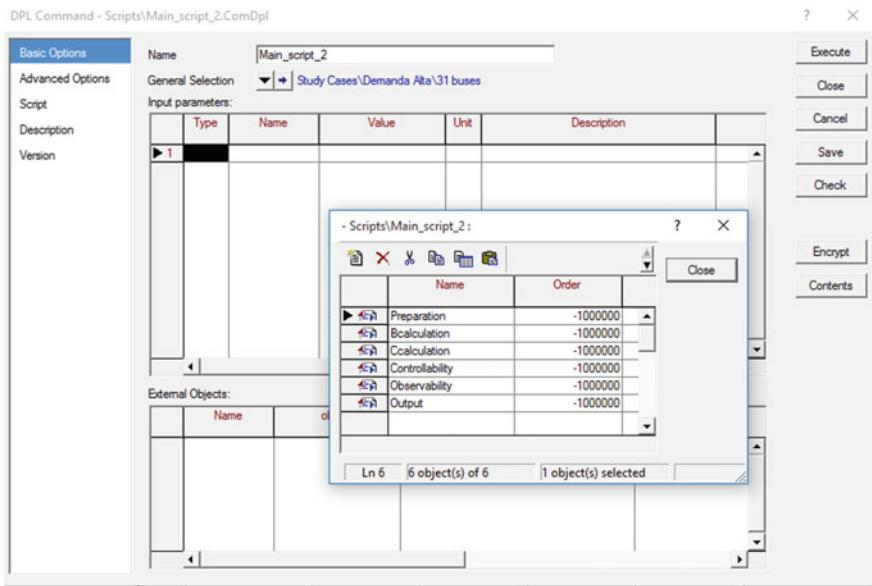


Fig. 7.3 DPL Main script_2 screenshot

```
## Main_script_2 ##
string path;
set StudyCases;
object Folder,Case,Initial,Modal;
!Preparation for the simulation
path='D:\Temp';                                !Set the path of the target folder
Preparation:path=path;
Bcalculation:path=path;
Ccalculation:path=path;
Controllability:path=path;
Observability:path=path;
Output:path=path;

Preparation.Execute();!Get the states variables for calculation
Bcalculation.Execute();!Get matrix B
Ccalculation.Execute();!Get matrix C
Controllability.Execute();!Get mode controllabilities
Observability.Execute();!Get mode observabilities
Output.Execute();!Write the output files
```

Fig. 7.4 DPL for preparation, calculations and exporting results

the “VariableToIdx_Amat.txt” file to get the state variables. For instance, consider the first three rows of the following “VariableToIdx_Amat.txt” file.

1. SING\BESS\PWM BESS ANG.ElmVscmono “phiu”
2. SING\BESS\BESS Angamos\Energy.ElmDsl “xE”
3. SING\BESS\BESSAngamos\FrequencyControl.ElmDsl “xdpdt”

Here, the first element of each line corresponds to the variable number. Thus, the number in the last line of the file should coincide with the size of the A matrix. Although not used for calculation, this number allows to identify each state variable. The second part of each line is the DIgSILENT path of the element, which is related to the corresponding state variable. This path is used to search the element of interest with a filter (SetFilt*) and assign it to a new *IntMon** object in All calculation. Finally, the third part in quotation marks gives always the name of the state, which is used as the selected variable of the *IntMon** object. Figure 7.5 shows the subroutine *Preparation*.

7.3.1 State-Space Representation: Matrix B and C

Contrary to matrix A , matrix B and C are not unique and depend on the user definition. For the following programs, matrix B is calculated assuming that the input variables correspond to active power injections at the buses of the general selection (SetSelect*)—damping regulating devices are considered in this chapter, which modulate active power. Similarly, and for the sake of simplicity, direct observation of all machine speeds is considered as possible feedback signals.

Consider the system subject to a single power variation Δu_i . Thus,

$$\Delta \dot{x} = A \Delta x + B_i \Delta u_i \quad (7.25)$$

Assume this input is a unit step function; thus, $\Delta x(t) = 0, \forall t \leq 0$, and the unit step is suddenly applied at $t = 0$. Using Forward Euler formula, the following expression is obtained after the first time step (Δt):

$$\Delta \dot{x}(\Delta t) \approx A \underbrace{\Delta x(0)}_{=0} + B_i \underbrace{\Delta u_i(0)}_{=1} \quad (7.26)$$

$$\Rightarrow B_i \approx \Delta \dot{x}(\Delta t) \approx \frac{\Delta x(\Delta t) - \overbrace{\Delta x(0)}^{=0}}{\Delta t} = \frac{x(\Delta t) - x(0)}{\Delta t} \quad (7.27)$$

Therefore, the column i of the matrix B is estimated by the application of this unit step function; small time steps are required, so the error of the Forward Euler equation is minimal. Based on the superposition principle, all columns of matrix B are determined by repeating this procedure for all buses of the general selection (SetSelect*). Figure 7.6 represents the B matrix calculation flow chart.

Figure 7.7 shows a screenshot of the *Bcalculation* element dialog and content tab. The same general selection (SetSelc*) used in *Main_script_2* must be selected. Additionally, a load (ElmLod*) call “Mobile Load” of 100 MW (1 pu) is used for the load step and selected as an external object. The input parameter “s”

```

!## Preparation ##
int ierr,count,i1,i2,a,iErr;
double index;
string var,str1,str2,s, file1;
set elements;
object element,data,Results;

a=0;
ierr=1;
count=1;
Filter:isubfold=1;
Results=GetCaseCommand('ElmRes');
file1='VariableToIdx_Amat.txt';
iErr=strcmp(path,'');
if(iErr==0){
    Error('No path selected. Select a path');
    exit();
}

iErr=strcmp(file1,'');
if(iErr==0){
    Error('No file name selected. Insert the name of the file');
    exit();
}
s=sprintf('%s\\%s',path,file1);
fopen (s,'r',0);

while (ierr>-1){
    if (count==1){
        ierr=fscanf(0, '%d',index);
        count=count+1;
    }
    if (count>1){
        while(count>1.and.count<4.and.ierr>-1){
            ierr=fscanfsep(0, '%s',path,'"',0);
            if (count<3){
                count=count+1;
                i1 = strstr(path, 'SING');!Replace by grid name
                str1 = strcpy(path,i1,120);
                i2 = strstr(str1, ' ');
                str2 = strcpy(str1,0,i2);
                Filter:objset=str2;
                elements=Filter.Get();
                element=elements.First();
                data=Results.CreateObject('IntMon');
                data:obj_id=element;
            }
            else {
                count=1;
                var=printf('s:%s',path);
                data.AddVar(var);
            }
        }
    }
}
fclose(0);

```

Fig. 7.5 DPL for adding state variables in all calculations

determines the path of the file “VariableToIdx_Amat.txt” and is assigned to the main program.

The script of *Bcalculation* is shown in Fig. 7.8. The first lines of code prepare the elements and set the simulation times to approximate the coefficients of matrix

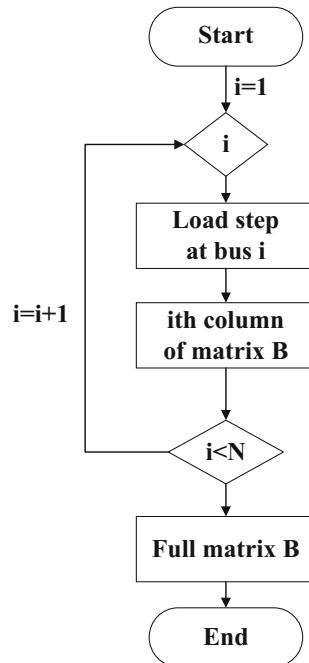


Fig. 7.6 Flow chart of matrix B calculation

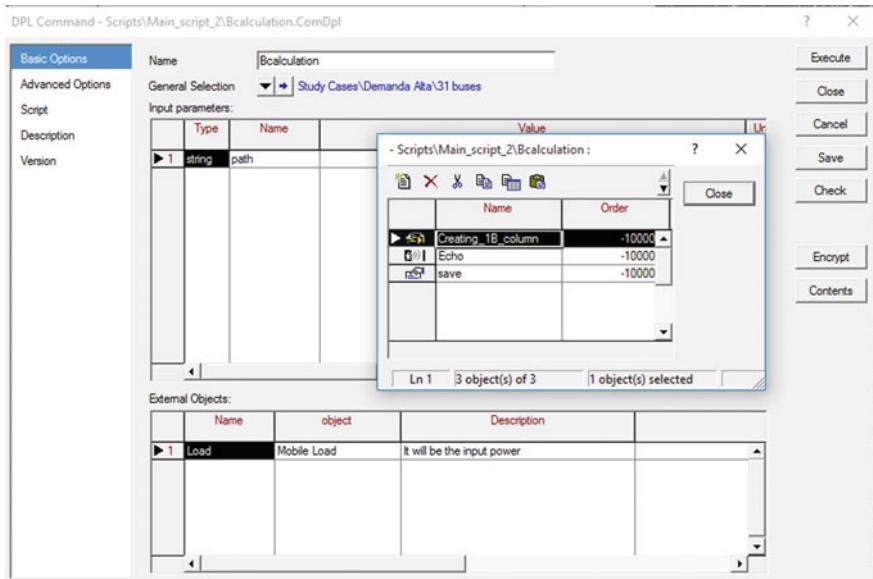


Fig. 7.7 “Bcalculation” element dialog and content tab

```

!## Bcalculation #
set terms,events;
object event, term, node, switch,Results,inicial,sim,inputs,MatrixB;
int aux, nin;
string path,str,input_description, s,file1;

Echo.Off();
ResetCalculation();
ClearOutput();

!Getting case elements
Results=GetCaseCommand('ElmRes');
file1='VariableToIdx_Amat.txt';
s=sprintf('%s\\%s',path,file1);
inicial=GetCaseCommand('ComInc');
sim=GetCaseCommand('ComSim');
inputs=GetCaseCommand('IntEvt');
MatrixB=GetCaseObject('MatrixB.IntMat');
MatrixB.Init(1,1);
Creating_1B_column:path=path;

!Preparing calculations
inicial:p_resvar=Results;
inicial:dtgrd=0.0001;
inicial:tstart=0;
inicial:dtout=0.0001;
inicial:p_event=inputs;
sim:tstop=0.0001;
events=inputs.GetContents();
event=events.First();                                !Must be created in active study case
event:time=0;
event:i_switch=1;
terms = SEL.GetAll('ElmTerm');                      !From general selection
terms.SortByName(0);
term = terms.First();

aux=1;
while (term) {
    if (term:outserv=0){
        node=term.CreateObject('StaCubic', 'CubiculoXX');
        node.RemoveBreaker();
        switch=node.AddBreaker();
        switch:on_off=0;                            !Initial state:open
        Load:bus1=node;
        event:p_target=Load;
        inicial.Execute();
        sim.Execute();
        printf('Input power %d in %s',aux,term);
        Creating_1B_column:column=aux;
        Creating_1B_column:s=s;
        Creating_1B_column.Execute();
        aux=aux+1;
        term = terms.Next();
        ResetCalculation();
        Delete(node);
    }
    else {
        term = terms.Next();
    }
}
input_description=sprintf("%s\\Input_description.out",path);

```

Fig. 7.8 DPL for calculating matrix B

```

save:f=input_description;
save.Execute();
ClearOutput();
printf('The program Bcalculation has finished, B has %d columns',aux-1);

```

Fig. 7.8 (continued)

B after a small time of 0.0001 s. In order to perform the load step, a single switch event (`EvtSwitch*`) must be manually created. The iteration starts alphabetically creating a node (`StaCubic*`) and a breaker added to each bus (`ElmTerm*`) of the general selection. This node then is assigned to the Mobile Load, which is selected for the switch event. Once the load is related to the bus of interest, the initial conditions (`ComInc*`) and simulation (`ComSim*`) are executed. An additional program called *Creating_1B_column* is executed to calculate the coefficients of B with the results from the simulation. The procedure is repeated for all the buses in the general selection. A file called “Input description.out” is generated to allow the user a fast identification of the input definitions.

The script of *Creating_1B_column* is shown in Fig. 7.9. This subroutine is in charge of the calculation of the coefficients of the i th column of B after the i th load step is applied. The parameter i is entered as an input (called “column”) of the “*Creating_1B_column*” program, assigned by code in *Bcalculation*. According to Eq. (7.27), the program needs to read the value of all state variables at $t = 0$ and at $t = \Delta t$ (one time step). This is done by reading the “VariableToIdx_Amat.txt” which gives the number of each state, the path of the element related to the state and the name of the state. The index gives the position in B_i related to that state, while the path and name are used to find the element object with a filter. The element together with the variable name allows us to search in the Results element the position of the desired state. Then, the position (named “col” in the script) is used to get the value of the state at both time of interests to calculate the coefficient and assign it to the respective position of matrix B .

Consider now the computation of matrix C . Since the output variables are defined as the direct speed observation of all machines, the i th row of this matrix is full of zeros except for one term with value “1” associated with the speed of the i th machine. The program *Ccalculation* is used to generate this matrix. The input parameter “path” is set in the program *Main_cript_2* and allows to read the file “VariableToIdx_Amat.txt” which gives the index of each machine speed variable to save them in matrix *Cindex*. This matrix is then used to build matrix C . Figure 7.10 shows the script of *Ccalculation*. Additionally, the file “Output_description.txt” is generated to allow the user an easy access to the output definition.

```

!## Creating_1B_column ##
int ierr,count,i1,i2,col,nval,cont,row;
double index,aux_ini,aux_end,aux,ti,tf,txx;
string var,str1,str2,s, Dpath;
set elements;
object element,MatrixB,Results;

ierr=1;
count=1;
Filter:isubfold=1;
Results=GetCaseObject('ElmRes');
LoadResData(Results);
col=ResIndex(Results,Results,'b:tnow');
nval=ResNVal(Results,0);                                !Return the number of values

txx=0;
cont=1;
tf=time();
while (txx==0){
    GetResData(ti,Results,nval-cont,col);
    txx=tf-ti;
    row=nval-cont;
    cont=cont+1;
}
MatrixB=GetCaseObject('MatrixB.IntMat');
fopen (s,'r',0);
while (ierr>-1){
    if (count==1){
        ierr=fscanf(0,"%d",index);
        count=count+1;
    }
    if (count>1){
        while(count>1.and.count<4.and.ierr>-1){
            ierr=fscanfsep(0,"%s",Dpath,'"',0);
            if (count<3){
                count=count+1;
                i1 = strstr(Dpath,'SING'); !Replace by grid name
                str1 = strcpy(Dpath,i1,120);
                i2 = strstr(str1,' ');
                str2 = strcpy(str1,0,i2);
                Filter:objset=str2;
                elements=Filter.Get();
                element=elements.First();
            }
            else {
                count=1;
                var=sprintf('s:%s',Dpath);
                col=ResIndex(Results,element,var);
                GetResData(aux_ini,Results,row,col);
                GetResData(aux_end,Results,nval-1,col);
                aux=(aux_end-aux_ini)/txx;
                MatrixB.Set(index,column,aux);
                Results.Flush();
            }
        }
    }
    fclose(0);
    ReleaseResData(Results);
}

```

Fig. 7.9 DPL for creating a column of matrix B

```

## Calculation ##
int ierr,count,i1,i2,comp,a,m,row,col;
double index,aux;
string var,str1,str2,output_description,s,file1, s2,file2;
set elements;
object element, Cindex,MatrixC;

ierr=1;
count=1;
aux=1;
file1='VariableToIdx_Amat.txt';
file2='Observability description.txt';
s=sprintf('%s\\%s',path,file1);
s2=sprintf('%s\\%s',path,file2);
Cindex=GetCaseCommand('Cindex.IntMat');
Cindex.Init(1,1);

fopen (s,'r',0);
fopen (s2,'w',1);
while (ierr>-1){
    if (count=1){
        ierr=fscanf(0,"%d",index);
        count=count+1;
    }
    if (count>1){
        while(count>1.and.count<4.and(ierr>-1){
            ierr=fscanfsep(0,"%s",path,"",0);
            if (count<3){
                count=count+1;
                i1 = strstr(path, 'SING'); !Replace by grid name
                str1 = strcpy(path,i1,120);
                i2 = strstr(str1, ' ');
                str2 = strcpy(str1,0,i2);
                }
            else {
                count=1;
                comp=strcmp(path, 'speed');
                if (comp=0){
                    Cindex.Set(aux,1,index);
                    fprintf(1,'Output %d is state variable number %d from
%s',aux,index,str2);
                    aux=aux+1;
                }
            }
        }
    }
    fclose(1);
    fclose(0);
MatrixC=GetCaseCommand('MatrixC.IntMat');
row=Cindex.NRow();
col=index-2;
MatrixC.Init(row,col);

for(m=1;m<row+1;m=m+1){
    a=Cindex.Get(m,1);
    MatrixC.Set(m,a,1);
}
printf('The program Ccalculation has finished, c has %d columns',aux-1);

```

Fig. 7.10 DPL for getting matrix C

7.3.2 Mode Controllability and Observability

Mode controllability is calculated using the matrix B and the left eigenvector related to the eigenvalue of interest. To obtain this eigenvector, the index of the respective eigenvalue is needed. This is obtained by reading the “Evals.mtl” file generated from *Main_script_1* and by filtering the modes using the previous information from a Modal Analysis calculation. In this code, only the imaginary part is used to select the eigenvalue, but a more restrictive searching (in the case of several modes in the same frequency range) could be achieved by adding a filter to the real part as well. After getting the mode index, the program proceeds to read the file “IEV.mtl” saving in two matrices the real and imaginary parts of the left eigenvector of interest. These values are used to get the magnitude of the mode controllability indices for each location, which are saved in the matrix *Mctr.IntMat*. Figure 7.11 shows the subroutine *Controllability*.

For the mode observability, the calculation is similar by using the matrix C and the right eigenvector. Since the rows of C contain only one nonzero element, the matrix multiplication gives one element of the right eigenvector (in the position of the nonzero element) for each output. Then, the calculation is simplified using the matrix *Cindex*. Figure 7.12 shows the subroutine *Observability*.

The final step consists on normalising the mode controllability and observability on their higher values to obtain *CI* and *OI*. The indices are written in two separate output csv file. Figure 7.13 shows the subroutine *Output* which performs the described task.

7.3.3 Guideline to Execute the Program for Any Arbitrary System

- Choose your working path and assign it to the variable *path* in the program *Main_script_1* and *Main_script_2*.
- Perform modal analysis calculation to determine the mode of interest. If needed, modify the search range for the imaginary part of the eigenvalue of interest to the most suitable range for your system. This change is done in the programs *Controllability* and *Observability*.
- Create a constant power load of 100 MW (base power in DIgSILENT by default) and select it as an external object in program *Bcalculation*.
- Create a switch event in the active study case.
- Create a general selection in the active study case with the prospective installation locations and select it in program *Main_script_2* and *Bcalculation*.
- Replace the string “SING” by the name of your grid (ElmNet)—Typically “Grid”—in programs *Preparation*, *Bcalculation* and *Ccalculation*.

```

## Controllability ##
int x,y,ierr,count,index,n,e1,e2,row,col,nc,ni,conte1,conte2,mode;
double real,imag,aux,b,ar,ai,sumr,sumi,magnitude,ctrl;
string lEV,EVals,PartFacs,path,file1,file2,file3;
object Mreal,Mimag,Mindex,B,Mctrl;

!Get the path
file1='lEV.mtl';
file2='EVals.mtl';
file3='PartFacs.mtl';

lEV=sprintf('%s\\%s',path,file1);
EVals=sprintf('%s\\%s',path,file2);
PartFacs=sprintf('%s\\%s',path,file3);
!Get the matrix

Mreal=GetCaseCommand('Mreal.IntMat');
Mimag=GetCaseCommand('Mimag.IntMat');
Mindex=GetCaseCommand('Mindex.IntMat');
B=GetCaseCommand('MatrixB.IntMat');
Mctrl=GetCaseCommand('Mctrl.IntMat');
Mreal.Init(1,1);
Mimag.Init(1,1);
Mindex.Init(1,1);
Mctrl.Init(1,1);
count=1;
ierr=1;
n=1;

!Determine the index of eigenvalue of interest
fopen (EVals,'r',0);
while (ierr>-1){
    if (count=1){
        ierr=fscanf(0, '%d',index);
        count=2;
    }
    if (count<4){
        ierr=fscanf(0, '%d',aux);
        count=count+1;
    }
    if (count=4){
        ierr=fscanf(0, '%d',imag);
        count=1;
        if (imag>2.1 .and.imag<2.3){ !mode filter
            Mindex.Set(n,1,index);
            n=n+1;
        }
    }
}
fclose(0);

mode=Mindex.Get(1,1);!Get the mode of interest

!Getting the left eigenvector asociated with the eigenvalue of interest
fopen (lEV,'r',0);
count=1;
ierr=1;
while (ierr>-1){
    if (count=1){
        ierr=fscanf(0, '%d',x);
        count=2;
    }
    if (count=2){

```

Fig. 7.11 DPL for obtaining mode controllability

```

        ierr=fscanf(0, '%d',y);
        count=3;
    }
    if (count=3){
        ierr=fscanf(0, '%d',real);
        count=4;
    }
    if (count=4){
        ierr=fscanf(0, '%d',imag);
        count=1;
    }
    if (y==mode){
        Mreal.Set(1,x,real);      !Left eigenvector for mode y (real part)
        Mimag.Set(1,x,imag);     !Left eigenvector for mode y (imaginary part)
    }
}
fclose(0);
nc=Mreal.NCol();
ni=B.NCol();
for (col=1;col<(ni+1);col=col+1){
    sumr=0;
    sumi=0;
    for (row=1;row<(nc+1);row=row+1){
        ar=Mreal.Get(1,row);
        ai=Mimag.Get(1,row);
        b=B.Get(row,col);
        sumr=sumr+ar*b;
        sumi=sumi+ai*b;
    }
    ctrl=sqrt(sumr*sumr+sumi*sumi);
    Mctrl.Set(1,col,ctrl);
}
printf('The program Controllability has finished');

```

Fig. 7.11 (continued)

- Execute *Main_script_1* and manually delete header line and any other description line without index, path and variable within the body of the file *VariableToIdx_Amat.txt*. For verification purposes, remember that the number of lines in the file should be the same as the number of state variables.
- Execute *Main_script_2*.

7.4 Real System Application: The Northern Chile Interconnected System

The Northern Chile Interconnected System is a very particular system that, as to 2015, had an installed capacity of 4150 MW, maximum demand around 2400 MW and an inertia of about 3.86 s base on the installed power. The NCIS has been connected to the Argentinian Interconnected System (AIS) since 2015, improving its operational performance in aspects such as frequency quality and voltage profiles. However, without proper control actions, the appearance of a critical inter-area oscillation between both systems is imminent. Hence, the use of damping regulating

```

!## Observability ##
int x,y,ierr,count,index,n,e1,e2,row,col,nc,ni,conte1,conte2,mode,rin;
double real,imag,aux,c,ar,ai,sumr,sumi,magnitude,obser;
string rEV,EVals,PartFacs, path,file1,file2,file3;
object Mreal,Mimag,Mindex,C,Mobser;

!Get the path      file1='rEV.mtl';
file2='EVals.mtl';
file3='PartFacs.mtl';

rEV=sprintf('%s\\%s',path,file1);
EVals=sprintf('%s\\%s',path,file2);
PartFacs=sprintf ('%s\\%s',path,file3);

!Get the matrix
Mreal=GetCaseCommand('Mreal.IntMat');
Mimag=GetCaseCommand('Mimag.IntMat');
Mindex=GetCaseCommand('Mindex.IntMat');
C=GetCaseCommand('Cindex.IntMat');
Mobser=GetCaseCommand('Mobser.IntMat');

Mreal.Init(1,1);
Mimag.Init(1,1);
Mindex.Init(1,1);
Mobser.Init(1,1);
count=1;
ierr=1;
n=1;

!Determine the index of eigenvalue of interest
fopen (EVals,'r',0);
while (ierr>-1){
    if (count=1){
        ierr=fscanf(0, '%d',index);
        count=2;
    }
    if (count<4){
        ierr=fscanf(0, '%d',aux);
        count=count+1;
    }
    if (count=4){
        ierr=fscanf(0, '%d',imag);
        count=1;
        if (imag>2.1 .and.imag<2.3){!mode filter
            Mindex.Set(n,1,index);
            n=n+1;
        }
    }
}
fclose(0);
mode=Mindex.Get(1,1);!Get the mode of interest
!Getting the right eigenvector asociated with the eigenvalue of interest
fopen (rEV, 'r',0);
count=1;
ierr=1;

while (ierr>-1){
    if (count=1){
        ierr=fscanf(0, '%d',x);
        count=2;
    }
    if (count=2){
        ierr=fscanf(0, '%d',y);
        count=3;
    }
}

```

Fig. 7.12 DPL for obtaining mode observability

```

        }
        if (count=3){
            ierr=fscanf(0, '%d', real);
            count=4;
        }
        if (count=4){
            ierr=fscanf(0, '%d', imag);
            count=1;
        }
        if (y==mode){
            Mreal.Set(1,x,real);      !Right eigenvector for mode y (real part)
            Mimag.Set(1,x,imag);     !Right eigenvector for mode y (imaginary part)
        }
    }
fclose(0);

!Calculating the observability vector
nc=C.NRow();
for (row=1;row<(nc+1);row=row+1){
    rin=C.Get(row,1);
    ar=Mreal.Get(1,rin);
    ai=Mimag.Get(1,rin);
    obser=sqrt(ar*ar+ai*ai);
    Mobser.Set(1,row,obser);
}
printf('The program Observability has finished');

```

Fig. 7.12 (continued)

devices, such as ESS, through active power injection is attractive to solve this problem [7, 8]. In order to decide the ESS installation location, thirty-one bus candidates in the NCIS are selected and evaluated using *CI*. Similarly, assuming ideal wide-area measurements, twenty feedback signals (machine speed) are proposed and analysed with *OI*.

In a conventional project analysis, the pair location/signal with higher *CI* and *OI* is tested by performing modal analysis and time-domain simulations when the damping regulating device is connected. For the sake of clarity, this analysis is separated to see the independent effects of each decision. Both location and signal are compared for the worst, the best and one intermedium candidate.

Although any ESS can be employed, e.g. flywheels, ultracapacitors, others, for simplicity, the Battery Energy Storage (BES) model from the DIgSILENT library is used. The template used is “BatteryWithFrequencyControl_10kV_30MVA” for which the nominal power has been changed to 50 MW; the dead band is neglected and the droop gain of the frequency control is set to 0.002. Additionally, a phase compensator is included to the frequency control loop to add the necessary angle compensation for the inter-area mode.

7.4.1 Controllability Analysis

The results of the controllability index for the thirty-one prospective locations obtained from the file CI.csv are plotted in the NCIS map and shown in Fig. 7.14.

```

## Output ##
double m,m_max;
int a,i,a1,a2;
string file_c,file_o,path,file1,file2;
object Mctrl,Mobser;

file1='CI.csv';
file2='OI.csv';
file_c=sprintf('%s\\%s',path,file1);
file_o=sprintf('%s\\%s',path,file2);
Mctrl=GetCaseCommand('Mctrl.IntMat');
Mobser=GetCaseCommand('Mobser.IntMat');
a1=Mctrl.NCol();
a2=Mobser.NCol();
m_max=0;
for(i=1;i<a1+1;i=i+1){
    m=Mctrl.Get(1,i);
    if (m>m_max){
        m_max=m;
    }
}
fopen(file_c,'w',0);
for(i=1;i<a1+1;i=i+1){
    m=Mctrl.Get(1,i);
    m=m/m_max;
    fprintf(0,"%f",m);
}
fclose(0);
for(i=1;i<a2+1;i=i+1){
    m=Mobser.Get(1,i);
    if (m>m_max){
        m_max=m;
    }
}
fopen(file_o,'w',0);
for(i=1;i<a2+1;i=i+1){
    m=Mobser.Get(1,i);
    m=m/m_max;
    fprintf(0,"%f",m);
}
fclose(0);
printf('The program Output has finished'

```

Fig. 7.13 DPL for writing the output files

This map shows a pattern where the *CI* increases in the north direction as the buses get away from the interconnection with the AIS.

In order to compare the effect of the installation location alone, three different BES locations for a common feedback signal (CTTAR) are analysed. The *CI* and results from Modal analysis are shown in Table 7.1. The base case is defined as the case without BES.

The improvement of the damping ratio, which gives a measurement of the mode displacement, is consistent with the *CI*. In addition, a time-domain simulation after a 10 ms three-phase short-circuit in bus Crucero is performed to compare the effects of bus location Andes and Parinacota. Again, the results show satisfactory correlation (Fig. 7.15).

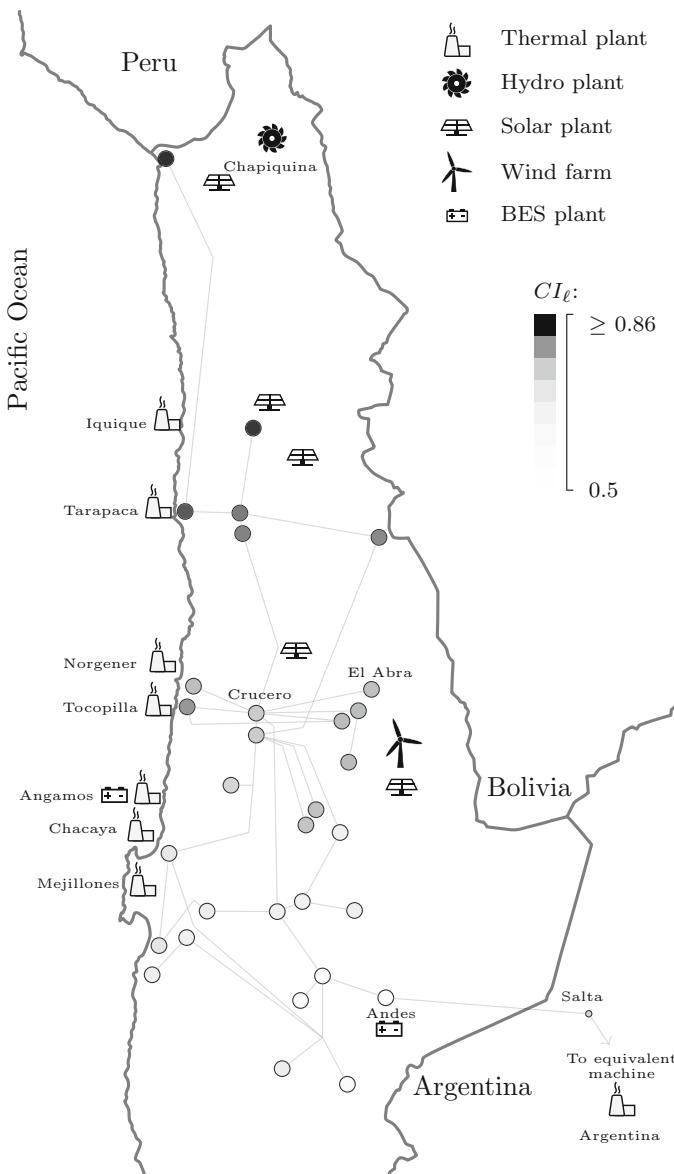
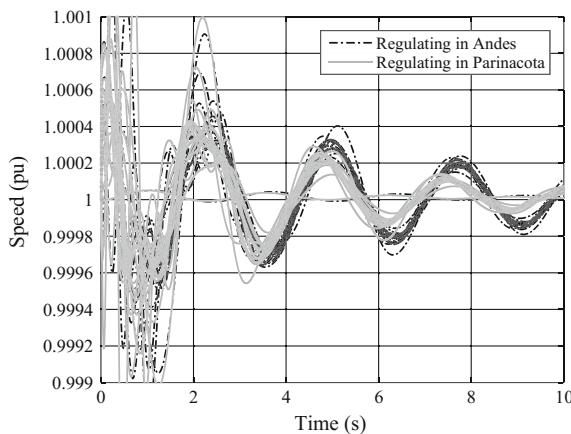


Fig. 7.14 CI results in the NCIS

Table 7.1 Installation location comparison

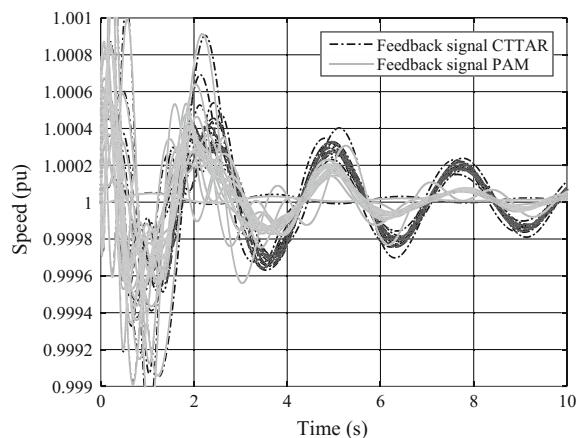
Bus location	<i>CI</i>	Eigenvalue	Damping ratio (%)
Base case	–	$-0.006 + j2.292$	0.27
Andes	0.54	$-0.144 + j2.292$	6.27
El Abra	0.80	$-0.235 + j2.291$	10.18
Parinacota	1	$-0.281 + j2.347$	11.88

**Fig. 7.15** Machine speeds for different BES locations**Table 7.2** Observability index for NCIS-AIS

Plant	Machine	<i>OI</i>
Tarapaca	CTAR	0.599666
Chacaya	CTM1	0.892592
	CTM2	0.775869
	CTM3-TG	0.717479
	CTM3-TV	0.701134
	CTH	0.721863
	CTA	0.721715
Norgener	NTO1	0.764271
	NTO2	0.764652
Mejillones	PAM	1
Tocopilla	U12	0.863075
	U13	0.861834
	U14	0.879836
	U15	0.852629
	U16	0.811950
Angamos	ANG1	0.682818
	ANG2	0.683066
Iquique	CAVA	0.722833
Chapiquina	CHAP	0.759904
Argentina	Argentina	0.062950

Table 7.3 Feedback signal comparison

Feedback signal	<i>OI</i>	<i>Ta</i>	<i>Tb</i>	Eigenvalue	Damping ratio (%)
Base case	–	–	–	$-0.006 + j2.292$	0.27
CTTAR	0.599666	0.4204	0.4528	$-0.144 + j2.292$	6.27
ANG2	0.683066	0.3094	0.6154	$-0.229 + j2.280$	9.99
PAM	1	0.3594	0.5298	$-0.292 + j2.272$	12.76

Fig. 7.16 Machine speeds for different feedback signals

7.4.2 Observability Analysis

The results of the observability index are shown in Table 7.2. The results show that the machine speeds in the NCIS have in general good observability, while the speed of the equivalent machine in the AIS is not a good option as a feedback signal.

A detailed comparison of three feedback signals, and the parameters of the phase compensator when the BES is connected in bus Andes, is shown in Table 7.3. The base case is defined as the case without BES.

Once again, the damping ratio improvement, as a measurement of the mode displacement, is consistent with the magnitude of the *OI*. A time-domain simulation after a 10 ms three-phase short-circuit in bus Crucero is performed to compare the effects of the feedback signals CTTAR and PAM when the BES is connected in bus Andes. The dynamic simulation is shown in Fig. 7.16. The results show agreement with the eigenvalue sensitivity approach.

7.4.3 Final Remarks

The proposed algorithms in this chapter proved to be effective to give a comparison index when a large number of installation locations and feedback signals are

considered in a real system. Thus, the effects of the damping regulating devices considering the best location/signal candidates—a small subset of the original analysis set—are evaluated with modal analysis and time-domain simulations for final selection. Note that, despite using the worse installation location in Table 7.2, the damping ratio can still be improved by selecting the appropriate feedback signal. For a more realistic analysis, this work can be improved by adding communication delays and frequency measurement units that can obtain frequency at any bus. Still, the scripts and applications presented in this document set the basis for additional development, including similar approaches that make use of state-space representation for real large-scale systems.

7.5 Conclusions

An approach to determine wide-area signals and installation locations of damping regulating devices using eigenvalue sensitivity has been explained and programmed in DPL. A state-space representation is estimated from dynamic simulations and used in addition to controllability and observability indices to reduce the number of location/signal. The effects of choosing the reduced set of location/signal candidates, when a damping regulating device is connected, are assessed with modal analysis and time-domain simulations. The approach is applied to analyse the critical inter-area oscillations between NCIS and AIS, and the independent effects of location and signal are studied for the connection of a BES in the NCIS. The indices are proven to be effective indicating the location/signal which drives the system from an almost unstable operation to conditions with a damping ratio greater than 11%. All programs provided in this chapter are generic and set the basis for further development in control and analysis of power systems using state-space representation.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 1509114. This work also made use of Engineering Research Center shared facilities supported by the Engineering Research Center Program of the National Science Foundation and the Department of Energy under NSF Award No. EEC-1041877 and the CURENT Industry Partnership Program.

References

1. H.F. Wang, F.J. Swift, M. Li, Indices for selecting the best location of PSSs or FACTS-based stabilisers in multimachine power systems: a comparative study. *IEE Proc. Gener. Transm. Distrib.* **144**(2), 155–159 (1997)
2. Josep Morato, Thyge Knüppel, Jacob Østergaard, Residue-based evaluation of the use of wind power plants with full converter wind turbines for power oscillation damping control. *IEEE Trans. Sustain. Energy* **5**(1), 82–89 (2014)

3. N. Yang, Q. Liu, JD. McCalley, TCSC controller design for damping interarea oscillations. *IEEE Trans. Power Syst.* **13**(4), 1304–1310 (1998)
4. S. Pizarro-Gálvez, H. Pulgar-Painemal, V. Hinojosa-Mateus, in *Parameterized Modal Analysis Using DiGILENT Programming Language*. Power factory applications for power system analysis (Springer International Publishing, Switzerland, 2014), pp. 221–248
5. P Kundur, in *Power System Stability and Control*, vol. 7, ed. by N.J. Balu, M.G. Lauby (McGraw-hill, New York, 1994)
6. F.L. Pagola, I.J. Perez-Arriaga, G.C. Verghese, On sensitivities, residues and participations: applications to oscillatory stability analysis and control. *IEEE Trans. Power Syst.* **4**(1), 278–285 (1989)
7. H. Silva-Saravia, H. Pulgar-Painemal, J. Mauricio, Flywheel energy storage model, control and location for improving stability: the Chilean case. *IEEE Trans. Power Syst.* **32**(4), 3111–3119
8. H. Pulgar-Painemal, Y. Wang, H. Silva-Saravia, On inertia distribution, inter-area oscillations and location of electronically-interfaced resources. *IEEE Trans. Power Syst.* **PP(99)**, 1–1 (Early access article), doi:[10.1109/TPWRS.2017.2688921](https://doi.org/10.1109/TPWRS.2017.2688921)

Chapter 8

Dynamic Stability Improvement of Islanded Power Plant by Smart Power Management System—Principles, Descriptions and Scenarios



Ali Parizad and Hamid Khoshkho

Abstract During the last decades, insufficient investment along with an increase in power demand have caused power system operating points to get closer to their stability boundary. In this condition, the need to fast and precise assessment of system stability status and proper execution of remedial actions results in a tendency of dispatching centres towards special protection system (SPS). This system is a smart monitoring and control system which detects the predefined condition and automatically executes pre-specified remedial actions to improve system stability. Indeed, such system can reduce human faults and prevent economic loss. Power management system (PMS) is a kind of SPS which is usually implemented in industrial power plants to intelligently manage remote devices and perform required actions against system contingency, especially in islanding condition. PMS has significant functions such as load shedding/sharing, generation shedding, generation mode control and import/export control to the control voltage, frequency and active/reactive power of the system. In this chapter, PMS configuration and its major functions are explained, and its impact on system stability is analysed through detailed dynamic simulations in DIgSILENT PowerFactory software. In these simulations, DIgSILENT simulation language (DSL) is used to model turbine-governor, excitation system and signals.

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_8) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

A. Parizad (✉)

MAPNA Electrical & Control Company, Tehran, Iran
e-mail: ali.parizad@ieee.org

H. Khoshkho

Department of Electrical Engineering, Sahand University
of Technology, Sahand New Town, Tabriz, Iran
e-mail: khoshkho@sut.ac.ir

Keywords Blackout prevention • DIgSILENT simulation language
 Generation mode control • Import/export control • Load/generation shedding
 Power management system (PMS) • Power system stability • Special protection system

8.1 Introduction

Power system instability is one of the threats to power systems all over the world, and great efforts have been made to propose proper methods to assess (detect or predict) stability status of the system and to perform remedial actions to prevent instability [1]. If the power plant is disconnected from the grid, the operation of the islanded system may be interrupted because of the lack of generation resources. The non-critical loads have to be shed according to their priorities in order to prevent blackout event [2, 3]. Since system instability usually leads to huge economic loss, national dispatching centres prefer to use electrical power system management technology in power plant industry. Power management system (PMS) is one of the smartest technologies proposed to prevent system instability [4]. The term PMS was formerly used to describe arrangements for the automatic starting and stopping of electrical generators to meet the actual load requirements, but nowadays, it is implemented to a very wide range of control systems and is able to control system smartly [5, 6].

The PMS system plays a key role in intelligently performing automatic generation control, frequency/voltage control, Import/Export control, initiation of automatic generator synchronisation, automatic synchronisation of inter-tie feeder circuit breakers, rapid load/generation shedding under abnormal condition, secondary control and so on. Whereas an unwanted event causes power system instability, PMS is designed and implemented in a power plant (especially in islanding condition) to detect a disturbance, select the best function immediately and send a proper command to prevent system instability smartly [7, 8].

Figure 8.1 illustrates the differences between systems with/without PMS. In a system without PMS, controllers work individually and send required set points or commands to their own generator. In contrast, in the smart system, PMS gets required a signal from the entire network (globally) and then sends set points or commands to equipment, considering whole power plant conditions.

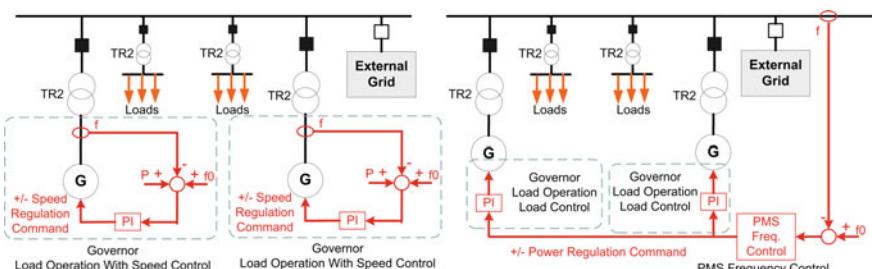


Fig. 8.1 Differences between systems with/without PMS

In [9], a scenario-based approach has been described to minimise the amount of load shedding to prevent system instability. Similarly in [10], load shedding is used to minimise the frequency swing in unwanted disturbances. It can be defined by a series of predefined operational scenarios. PMS components, such as automatic generation control, volt/VAR control, load shedding, etc. are discussed in [11]. Then related components are implemented in an industrial refinery to test the PMS system functionality.

In this chapter, some of the PMS applications such as load/generation shedding and load sharing are described. Although it has been described in [11] by some means, this chapter explains it differently in several aspects. First, generation mode control which is an important function in PMS is explained and implemented. The turbine-governor along with the changeover switch is simulated in DSL in order to change governor mode in disturbances rapidly. Second, to assess PMS application in DIgSILENT software, several scenarios are considered and studied in a test system. In these cases, DSL is implemented to simulate signals and system controllers such as AVR and governor. In [11], no simulation results are presented. Furthermore, the operator action is simulated; this way, the operator can correct frequency/power deviation manually when the system has reached its steady state stability. It provides a safe training method for operators while the systems remain undamaged.

An islanded power system encounters with different dynamics operational scenarios rather than those connected to a strong grid. To this purpose, different contingencies have been applied, and in each one, power plant stability is investigated and PMS functions are implemented, where required. This makes the critical scenarios to be detected in the power plant. By applying the smart PMS in the power plant, in the occurrence of the critical scenarios, the appropriate remedial actions such as changing set points, loads shedding and generation control will execute to prevent voltage/frequency instability. Also, it prevents sequential events which cause blackout.

8.2 Description and Implementation of a PMS

A power management system (PMS) is an integrated set of sensors, actuators, communication devices and networks, control logic and operator interfaces, which is used to provide real-time monitoring and smart control of the power system, especially in islanded condition, to improve its operation. To this purpose, different functions such as frequency, voltages and generation controls are continually executed by PMS to make system operation safe and efficient [6, 11]. Figure 8.2 shows a basic configuration of a PMS. In this structure, programmable logic controls (PLCs), where the PMS logic is implemented, are used to connect central control room (CCR) to the power plants and remote substations. So that required parameters such as frequency, voltage, active and reactive power and status of breakers are monitored through communication lines (e.g. fibre-optic cables).

Based on the above-mentioned monitored data, in CCR, the structure of the system is identified using PMS logic, and then the stability status and power quality of operating point are assessed. Also, automatic or non-automatic (i.e. using

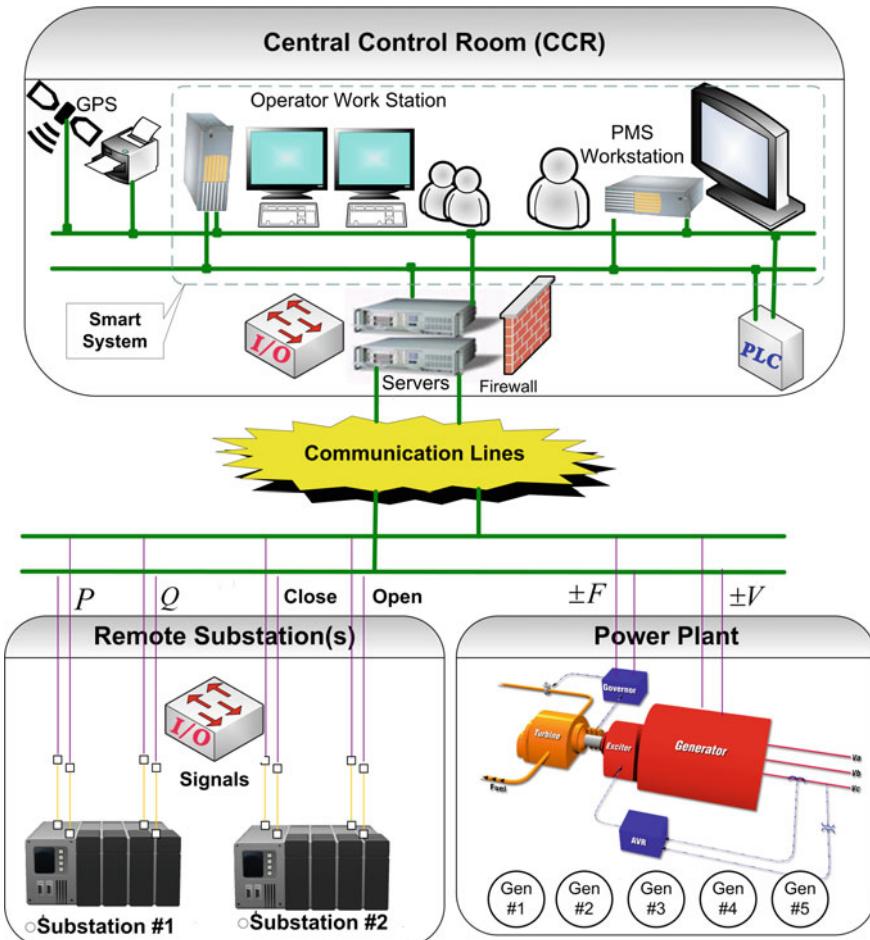


Fig. 8.2 Typical configuration of a PMS

operators' actions) remedial actions such as changing set points, loads shedding and generation control may be executed to improve system operation. Since the effectiveness of these remedial actions depends on their execution time, fast determination and execution of remedial actions are a matter of great importance.

8.2.1 Data Integrity Validation

All decisions made by PMS, for example, changing set points, are based on the data that have been collected from the various measuring devices of the power system. Since incorrectly measured data may result in the execution of inappropriate control actions and drive the power system to an abnormal operating condition, it is

essential that only valid data must be used. One of the most important functions of the PMS is data integrity validation, which is used to check validation of switches status and measured data. If discrepancies occur, an alarm is set to suspend those PMS functions which depend on the invalid data until the discrepancies exist. Consequently, data integrity validation enhances the robustness of the PMS and thus the power system [12]. In this project, critical analogue signals like active power are received from two different sources. For digital signals, a function block (FB) is written in PMS logic. In this function, double point input (DPI) structure is defined. In this strategy, if PMS receives 00 or 11, system detects it as faulty or indeterminate mode and no action will occur.

8.2.2 Work Station, Monitoring and Remote Control [12]

Workstation refers to hardware platform *designed* for transferring measured data from remote substations and power plants to central control room (CCR). As shown in Fig. 8.2, workstation usually includes PMS and operator workstations.

8.2.3 Input Signals

The following signals are required for PMS functionalities [12, 13]:

- Measured values (serial link): Generator/grid power (active and reactive), system frequency, load/feede power (active and reactive).
- Status information (serial link): Generator run/stop, breakers, load/feede circuit breakers, grid transformers, grid connectors and circuit breakers used to identify the islanding condition.
- Critical signals: Trip signals from the generator and generator incomer, trip signals from turbine-generator set (which will be treated as so-called pre-trip signals), under-frequency signals, frequency rate of change signals (df/dt).

8.3 Power Management Functions

Power system operators always play important roles to improve system operating point and prevent instability. However, operators' reactions to contingencies may be non-optimal especially when a major contingency occurs. Also, the operators' reaction is usually slow which may prevent the execution of timely control actions. In this regard, PMS can be used to assist operators in timely execution and optimum remedial actions to prevent instability [12].

8.3.1 Load Shedding Procedure in PMS

In power systems, excessive active power generation causes system frequency to increase and consequently may result in operation of frequency relays and system blackout occurrence. Also, some loads, like petrochemical loads, are highly sensitive to frequency deviation, and any under/over frequency may cause process failure and economic loss. Conventionally, to prevent these problems, load shedding is executed by frequency relays. Nowadays, the smart systems such as PMS provide a more intelligent tool able to select proper loads smartly and shed them in a fraction of a second to prevent instability. In PMS, there are three types of load shedding procedures which are described as the following [12, 14].

8.3.1.1 Fast Load Shedding

The fast load shedding is a fast response to a sudden major contingency in the network and used to keep the network alive smartly. The following signals must be transferred to the remote I/O modules of the controller, by the communication system, in order to detect contingencies:

- Trip signals from turbine-generator unit: This signal shows the failure of a generation unit and indicates that active/reactive power generation has become less than load demand and under frequency/voltage will be occurred.
- Switch status of the external grid: Detection of this signals shows that the plant has been disconnected from the grid, and under or over frequency condition will occur according to the direction of power exchange between the plant and grid before islanding. The solution is similar to generation trip.

Upon detection of such contingencies, the first shedding signal is sent from PMS to the load/feeder less than 75 ms after detection of a major contingency. As shown in Fig. 8.3, this delay consists of the reaction time of digital cards, load shedding calculations in PMS logic, propagation time and activation of the relay.

When a generating unit is tripped, the maximum power produced by this generator (including the active power and related spinning reserve) is lost and load shedding procedure should compensate the lost power. In this regard, the following

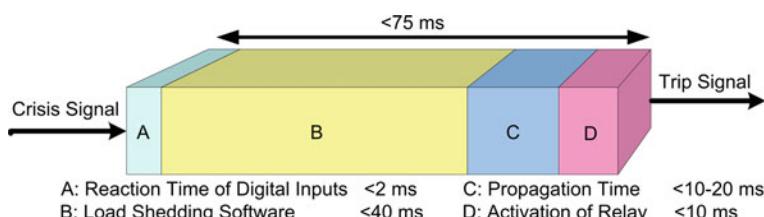


Fig. 8.3 Fast load shedding reaction time

relation may be used to calculate the amount of load demand which should be shed to prevent under-frequency condition [12, 14]:

$$P_{\text{ToBeShed}} = \pm P_{\text{Import/Export}} - (N_G \times P_{\text{Spin}}) + P_{\text{Offset}} \quad (8.1)$$

where $P_{\text{Import/Export}}$ is the active power exchange with the external grid (+ for import and—for export), N_G shows the number of the generator in running mode, P_{Spin} indicates the active power reserve of each generator and P_{Offset} is extra load which should be a shed to guarantee stability.

8.3.1.2 Under Frequency—Under Voltage Load Shedding

This procedure performs load shedding when frequency or voltage drops below a predefined threshold [12, 15].

8.3.1.3 Overload load shedding

This load shedding is executed when an overload condition occurs in the system.

8.3.1.4 Load Shedding List (LSL)

The LSL is the list of electrical consumers that the system, in the case of emergency, can disconnect from the electrical grid. A typical LSL is shown in Table 8.1.

As shown in the fourth column of this table, the priority is used to determine the order of execution of load shedding and can be changed only by a user with appropriate permissions. Here, those loads whose priority is 0 (red colour) should never be shed, and the ones with priority 1 are the first group of loads which will be shed in an emergency condition. In the fifth column, the current load consumption

Table 8.1 Example of load shedding list

No.	Group	Load Name	Priority	Load (MW)	Zone
1	---	Aromatics	1	12	A
2		Ammonia	0	13	A
3		Butanol	1	50	B
4	---	Alachlore	2	15	A
5		Alpha-Olefins	2	13	C
6		Butadiene	2	2	C
7	Group A	Urea Ammonia	1	23	D
		Vinyl Chloride		15	

acquired with a dedicated transducer is shown. The load shedding list can be divided into different groups. Consequently, the priority will be assigned to the first element of a group (Group A), and others inherit the same priority value.

8.3.2 Import/Export Control (I/EC)

Power import/export control ($P_{I/EC}$) is used to maintain the active power exchange with the grid at the desired value and can be set by the operator in pre-specified range ($P_{INmax} \leq P_{I/EC} \leq P_{OUTmax}$). P_{INmax} and P_{OUTmax} are maximum allowed exchanged power which is imported or exported. Figure 8.4 shows a block diagram of this controller where a PI controller causes generators to change their power generation. In this figure, P_{ni} is the rated power of the i th generator while P_n is the sum of the rated power of all generators connected under I/EC [12].

8.3.3 Generation Mode Control

PMS continually checks the system condition and, if necessary, changes operation mode of gas turbines to properly regulate system operating points and prevents instability (especially in islanded condition). In this subsection, various operation modes of gas turbines along with the PMS function (which is used to control generation modes of synchronous generators) will be described [14, 15].

- I. Start-up Mode: Normally, before connecting the generator to the grid, a turbine is in turning gear and rotates slowly (around 450–550 rpm). In start-up mode, operator command initiates synchronising sequence and causes fuel valves to open, and sufficient fuel (for starting the combustion) is led to the chamber and set the fire. Gradually, more fuel will be injected causing the turbine to reach the nominal speed, for example, 3000 rpm in 2 poles 50 Hz generator.
- II. Speed Control Mode: After start-up mode, the turbine has reached its rated speed, the control mode is automatically switched to speed control mode and synchronising sequence will be completed. Similar to start-up mode, speed control mode is carried out in local control mode, and remote control from PMS is not possible.

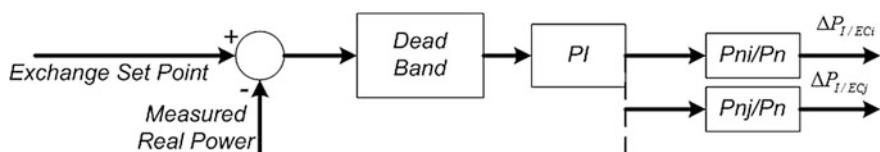


Fig. 8.4 Import/export control block diagram

III. Load Operation with Load Control (LOLC) Mode: In this condition, the main target is to control power generation; hence, governor control mode has to be switched to load operation with load control mode, and active power of generators regulates according to the frequency influence OFF or ON strategies. In the frequency influence OFF strategy, the control signal value (that will be sent to the PI controller) is directly obtained by comparing the produced power measured at machine terminals with the desired power set point (Fig. 8.5a). Thus, this strategy cannot automatically control the frequency of the system precisely. Against the first strategy, in the second strategy (i.e. with frequency influence ON), the frequency deviation is considered and active power of the generator is changed so that the frequency reaches around nominal frequency (Fig. 8.5b), and hence, it seems to be more appropriate. Nevertheless, frequency regulation by this strategy is slow, and if a severe disturbance occurs, the frequency may deviate considerably [12, 14].

In LOLC mode, remote control from PMS is possible, and hence, PMS logic can set the frequency influence to OFF or ON. Usually, to avoid unwanted stress on the machine given by a continuous regulation related to a too high sensitivity to frequency variations, the first strategy, i.e. frequency influence OFF, is selected. On the other hand, the second strategy is usually selected based on national dispatching centre request when the plant is disconnected from the grid (detected by PMS) to stabilise the island frequency automatically.

IV. Load Operation with Speed Control (LOSC) Mode: As mentioned before, although LOLC mode can regulate frequency, it is slow and cannot be used when a severe disturbance (which causes significant frequency deviation) occurs. To quickly achieve load generation balance and prevent significant frequency deviation, load operation with speed control mode may be used by PMS logic to regulate frequency rapidly.

PMS may switch governor mode into this control mode if:

- The plant disconnects from the grid (i.e. in islanded condition).
- A load rejection is detected in islanded condition.

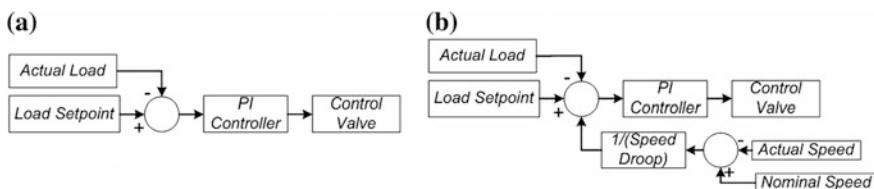
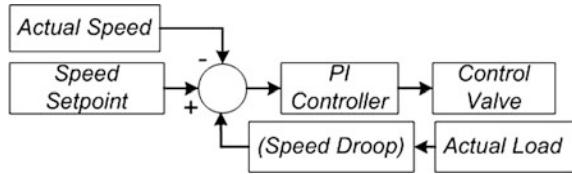


Fig. 8.5 Load control with frequency influence **a** OFF, **b** ON

Fig. 8.6 Block diagram of load operation with speed control mode



As shown in Fig. 8.6, in this mode, the regulation signal is determined by adding the difference between the measured speed and the set point value to the currently active power load weighed by the droop value. The control parameters are implemented in the governor logic, and operators can change only set points.

8.3.4 Generation Shedding Control (GSC)

When a power plant generates considerable power and exports power to the external grid, sudden disconnection of the plant from the grid will cause over frequency to occur. In this condition, it is necessary to implement generation shedding control (GSC) function in PMS, to regulate the frequency of islanded plant immediately. It should be stated that if LOSC mode exists in governor logic, the alternative approach is to shed fewer generators and change operating mode of some other generators to LOSC mode [12, 14, 15].

8.3.5 Secondary Frequency Control (SFC)

As mentioned in Sect. 8.3.3, although LOLC mode (with frequency influence ON or OFF) is the typical working mode of generators, it cannot precisely regulate frequency to the nominal value in steady state condition. To improve this condition, as shown in Fig. 8.7, secondary frequency control (SFC) is used.

The control block diagram of SFC controller is shown in Fig. 8.7. In this controller, the required active power change of each generator is calculated as:

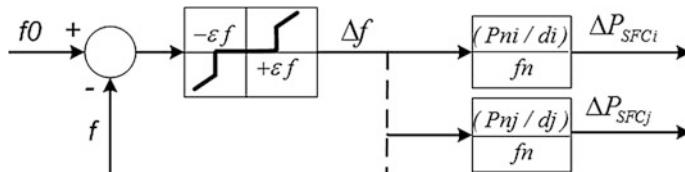


Fig. 8.7 Smart secondary frequency control block diagram

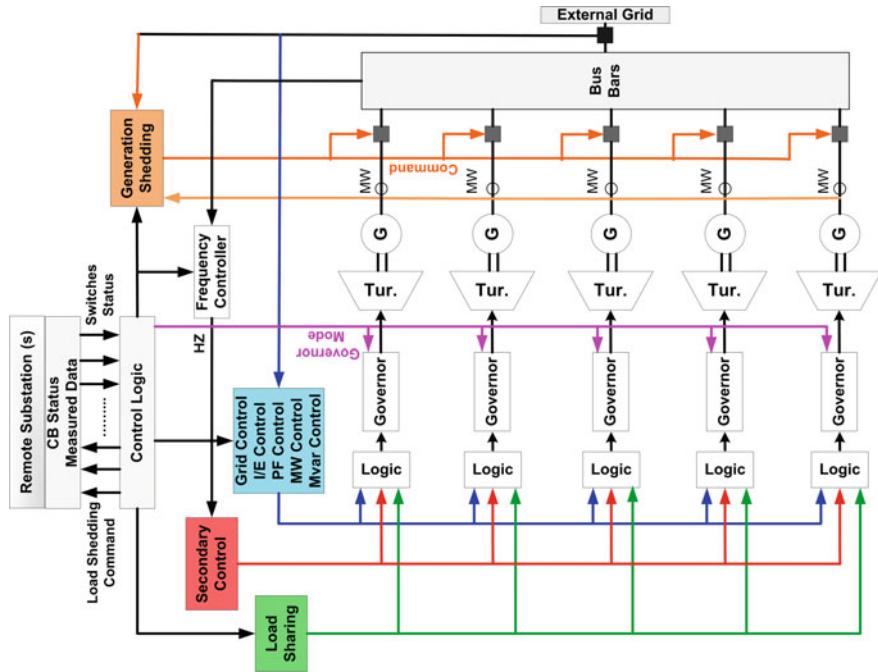


Fig. 8.8 Overall schematic of smart power management system controllers

$$\Delta P_i = \left(\frac{\Delta f}{f_n} \right) \times \left(\frac{\Delta P_{ni}}{d_i} \right) \quad (8.2)$$

where Δf is the frequency deviation from a nominal value, f_n is nominal frequency P_{ni} and d_i are the rated power and droop of i th generator, respectively [12]. The overall schematic of smart power management system controllers is shown in Fig. 8.8.

8.4 Power Management System Scheme

8.4.1 PMS Constraints

There are many constraints taken into account in the implementation of the power management system. The constraints considered in this study are as the following [12, 14].

- Power generation: It is defined according to the operation condition.

$$P_{\text{Gen}}^{\text{Min}} < P_{\text{Gen}} < P_{\text{Gen}}^{\text{Max}} \quad (8.3)$$

- Power import/export: It is defined according to the operation strategy and the lines capability.

$$P_{\text{Import}}^{\text{Max}} \leq 400 \text{ MW}, P_{\text{Export}}^{\text{Max}} \leq 400 \text{ MW} \quad (8.4)$$

- The maximum and minimum voltage limits.

$$V^{\text{Min}} < V < V^{\text{Max}}, V^{\text{Min}} = 0.94 \text{ p.u.}, V^{\text{Max}} = 1.05 \text{ p.u.} \quad (8.5)$$

- Frequency: It depends on the operation condition and relay settings. Also, the time period in which system works under this frequency is of importance.

$$F^{\text{Min}} < F < F^{\text{Max}} \quad (8.6)$$

Active power of loads, shedding signals, frequency set point and changeover signal in governor are the variables in hand to control the system stability in PMS.

8.4.2 PMS Control Algorithm

- (1) Power system monitoring (load flow in DIgSILENT) is continually executed to check the values (such as consumption, generation) and signal status of breakers and controllers.
- (2) The active power of loads and generators is continually calculated and sorted in the shedding list, according to the pre-settings.
- (3) According to the related logics, upon receiving the crisis signal, PMS calculates power imbalance considering the mentioned equation ($P_{\text{ToBeShed}} = \pm P_{\text{Import/Export}} - (N_G \times P_{\text{Spin}}) + P_{\text{Offset}}$), and the algorithm assesses different strategies as follows:
 - If the active power is imported from the grid and the islanding mode is occurred, PMS algorithm must calculate power imbalances, and if necessary, load shedding is executed.
 - If the active power is exported to the grid and the islanding mode is occurred, PMS algorithm must calculate power imbalances, and if necessary, generation shedding is executed. In this case, according to the excessive generation amount, change mode switch in governor logic (implemented in DSL) may be changed by PMS.
 - In the islanding mode, if the generators are tripped, PMS algorithm must calculate power imbalances, and if necessary, load shedding is executed.

- In the islanding mode, if the loads are tripped, PMS algorithm must calculate power imbalances, and if necessary, generation shedding is executed.
- (4) PMS sends the appropriate command to the related power system switches.
- (5) Finally, by compensating the imbalances between generation and consumption, the system remains stable. Consequently, PMS prevents a possible blackout.

8.5 Application of DIgSILENT to Model Smart PMS Operation

In this section, DIgSILENT PowerFactory software is used to execute time domain simulation to show the capability of PMS to maintain system stability. To this purpose, various contingencies are applied to test system, and it will be shown that how proper remedial actions carried out by PMS can prevent instability.

8.5.1 *Test System Description*

Figure 8.9 shows that in the base case condition, the system includes five generators and two 230/400 kV parallel transformers connecting the plant to the external grid. In this study, maximum capacity for these transformers is 200 MW and the maximum power exchange should be limited to 400 MW. Also, five 230/20 kV transformers are used to feed 26 petrochemical loads with a total consumption of 387 MW. Table 8.2 shows all loads and their priority level which have been used in load shedding procedure. As mentioned before, those loads with priority 0 should not be shed. In the following sub-subsection, gas turbine and excitation system model will be described and it will be shown that how a gas turbine operation mode (Sect. 8.3.3) will be changed.

8.5.2 *Dynamic Models for Transient Stability Study*

Dynamic models have to be simulated in DIgSILENT simulation language (DSL) in order to study the transient stability. Depending on the accuracy of the implemented models, the results can be varied. Therefore, a proper implementation of controllers is one of the important issues in the power system analysis. DigSILENT is a powerful software which can model different controllers by DSL. The following steps should be performed to achieve this purpose.

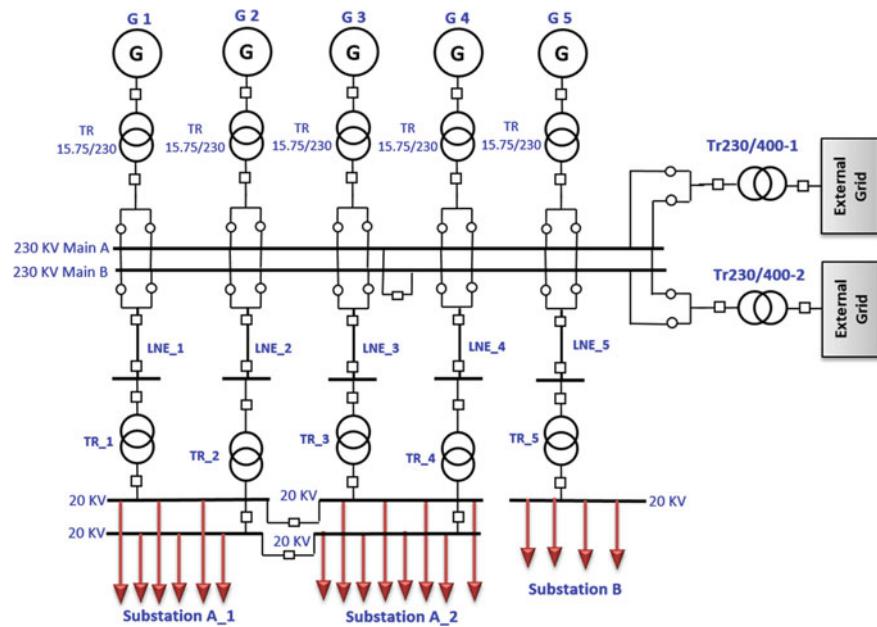


Fig. 8.9 Single-line diagram of the test system

Table 8.2 Priority list of loads

No.	Item	Load (MW)	Priority	No.	Item	Load (MW)	Priority
1	Alachlore	15	2	14	Future II	40	0
2	Alpha-Olefins	13	2	15	Future III	13	0
3	Ammonia	13	0	16	Gas Station	20	2
4	Aromatics	12	1	17	Harboun	12	1
5	Butadiene	2	2	18	Ethylhexanol	30	2
6	Butanol	50	1	19	Isopropyl Alcohol	12	0
7	Benzene	20	1	20	Methanol	12	1
8	Acid Acetic	5	2	21	Naphtha	2	1
9	Butenes	15	1	22	Olefin	2	2
10	Carbon Monoxide	2	1	23	Phenol	2	0
11	Ethylene	12	2	24	Polyethylene	12	1
12	Ethylene Oxide	30	1	25	Urea Ammonia	27	2
13	Future I	2	0	26	Vinyl Chloride	12	2

8.5.2.1 Composite Frame

In the first step, the composite frame which defines connections between the input and output signals of the different models is needed. In this study, four slots are implemented to connect the signals to the synchronous generator, excitation system, primary control unit and power system stabiliser (Fig. 8.10).

8.5.2.2 Composite Model

For each generator, it is necessary to define a composite generator model. This composite model (ElmComp) is used to link each slot in the generator frame to the related common model [12, 14].

8.5.2.3 Built-in Model

In the first slot of the composite model, the synchronous machine is used. This type of element, able to provide the input and output signals, is called a built-in model.

8.5.2.4 Model Definition

To define different control models such as excitation system, turbine-governor, model definition (BlkDef) is used. It is similar to the composite frame with differences in blocks and input and output variables.

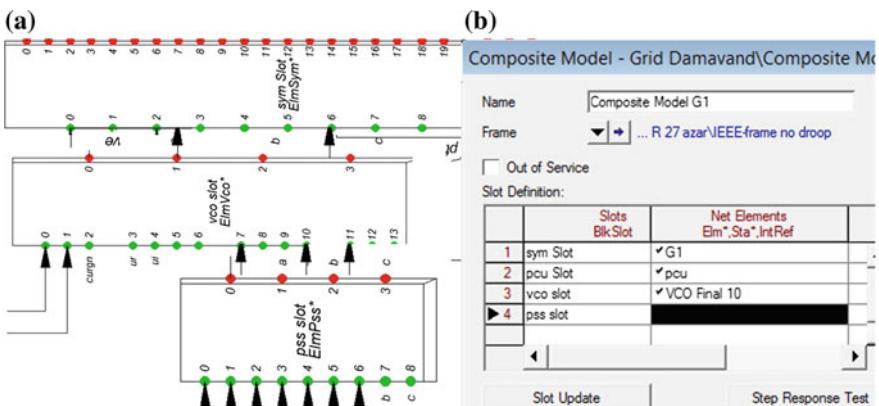


Fig. 8.10 a Composite generator frame in PowerFactory (BlkDef), b defined composite model in PowerFactory (ElmComp)

8.5.2.5 Common Model

Each system controller has a user-defined common model which combines a model definition (BlkDef) with the related parameter settings. The simulation of excitation system and turbine-governor model should be implemented in a common model in order to complete the composite model slots (PCU and VCO slot).

- Excitation System Dynamic Model: The excitation system is implemented in DSL to simulate automatic voltage regulation behaviour in the time domain simulations. The model is based on IEEE-ST [3] and some parts, and related parameters are optimised according to the real power plant.

Figure 8.11 shows the model definition of the excitation system with static excitation and field voltage feedback according to IEEE-ST [16].

This controller consists of a PI voltage regulator ($K_a(1 + 1/ST_a)$) with an inner loop field voltage regulator. The $(1 + 1/ST_r)$ function is used to consider the measurement delay. If the field voltage regulator is not implemented, the corresponding parameters K_f and K_{ss} are set to 0. VR_{max}/R_{min} represents the limits of the power rectifier. The power for the rectifier, V_{rsr} , may be supplied from the generator terminals or from an independent source. The parameters of this controller are defined in a common DSL model (ElmDsl) as shown in Fig. 8.12.

Initialization equations which show the excitation system model are written by considering that all the state variables satisfy $dx/dt = 0$. This consideration should be applied to each transfer function so that a set of algebraic equations can be solved. The excitation system initialization equations are as following:

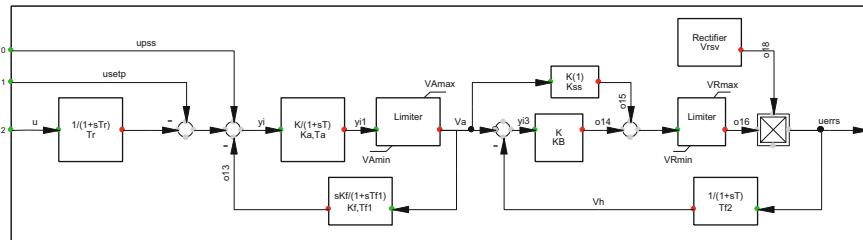


Fig. 8.11 Model definition of excitation system

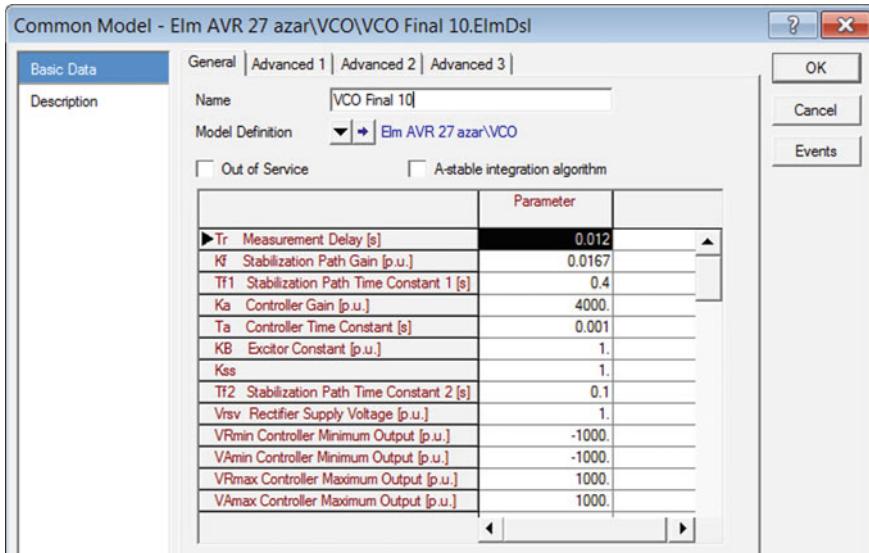


Fig. 8.12 Common model of excitation system

```

!The following equations are required for initial condition calculation
inc (xr)=u
inc (usetp)=u
inc (upss)=0
inc (xf1)=va
inc (xa)=va
inc (Va)=(1/Vrsv+KB)/(KB+Kss)*uerrs
inc (xf2)=uerrs
inc (Vs)=xa/Ka
!Definitions of parameters.
vardef (Tr)='s';'Measurement Delay'
vardef rdef(Ka)='p.u.';'Controller Gain'
vardef (Ta)='s';'Controller Time Constant'
vardef (VAmin)='p.u.';'Controller Minimum Output'
vardef (VAmax)='p.u.';'Controller Maximum Output'
vardef (VRmin)='p.u.';'Controller Minimum Output'
vardef (VRmax)='p.u.';'Controller Maximum Output'
vardef (Kf) ='p.u.';'Stabilization Path Gain'
vardef (Tf1)='s';'Stabilization Path Time Constant 1'
vardef (Tf2)='s';'Stabilization Path Time Constant 2'
vardef (Vrsv)='p.u.';'Rectifier Supply Voltage'
vardef (KB)='p.u.';'Excitor Constant'

```

- Turbine-Governor Dynamic Model: The simplified representation of gas turbines is shown in Fig. 8.13. It includes five controllers which are explained in detail as the following [14].
- Load Speed Controller: Load speed controller is the main control loop used to select a proper operation mode of the turbine (i.e. start-up mode, speed control

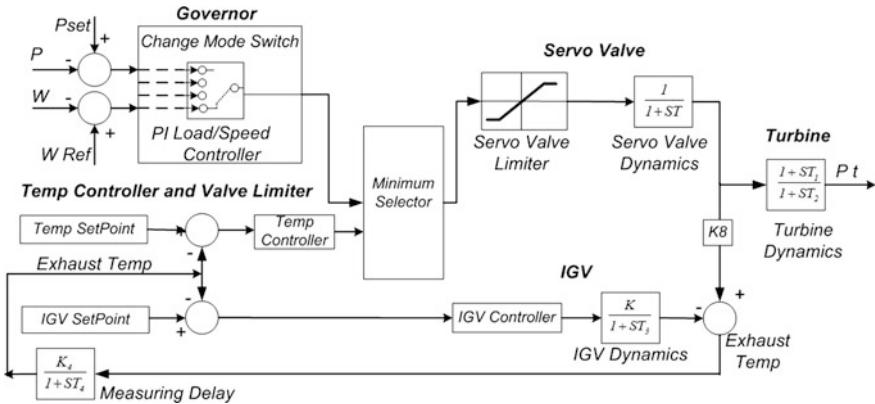


Fig. 8.13 Simplified representation of turbine-governor model

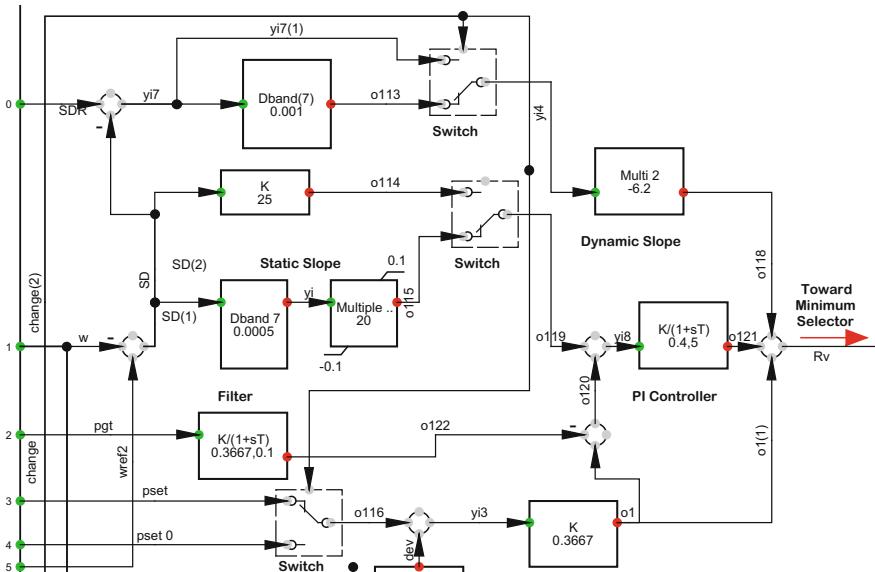


Fig. 8.14 Block definition of load speed controller implemented in DSL

mode, LOLC mode or LOSC mode introduced in Sect. 8.3.3) according to system condition. The inputs of this controller are speed and active power deviation from the set point values.

As shown in Fig. 8.14, the $K/(1+ST)$ function is used as the generator output power filter. Also, a PI Controller is defined for the load/speed control mode. The differences between the nominal and calculated frequency is an input signal for the

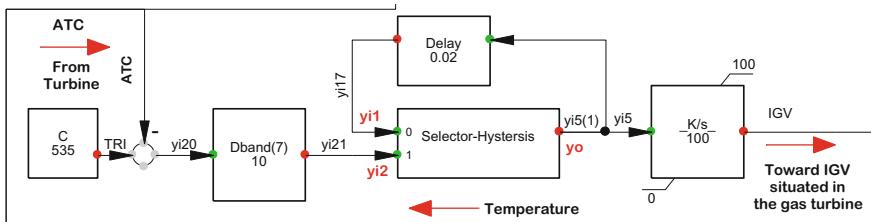


Fig. 8.15 Block definition of IGV controller implemented in DSL

dynamic and static slopes. Three switches are implemented to act whenever change of load/speed control mode in governor is needed.

- **IGV controller:** The action of inlet guide vanes (IGVs) can be considered in temperature controller. IGVs are situated in the air compressor stage of the gas turbine and can regulate the mass flow of air drawn into the compressor. Also, IGV directs the air into the compressor at the correct angle.

According to Fig. 8.15, the difference between the exhaust temperature signal (ATC), measured in turbine side, and IGV set point (i.e. 535 C^0) goes into the IGV controller. If the absolute value of the calculated difference is more than 10, it will pass through dead band block; else the output will be zero. The following DSL code implements this logic:

```
yo=select(abs(yi)>K,yi,0)
```

If the temperature rises, the IGV will be opened. In contrast, by reducing the temperature, the IGV is closed. This interaction is implemented in DSL by using a selector block by the following DSL code:

```
yo=select(yi1=0 .and. yi2<0,1,s1)
s1=select(yi1=1 .and. yi2<0,0,s2)
s2=select(yi1=0 .and. yi2>0,-1,s3)
s3=select(yi1=-1 .and. yi2>0,0,0)
```

Select code (`boolexpr, x, y`) returns `x` if `boolexpr` is true, else `y` will be returned. The results are sent to a PI controller (K/S) considering related limitations (i.e. 0–100) and then go towards IGV which is placed on the turbine side.

- **Temperature controller:** The output power of turbine increases in proportion to the load demand and consequently causes exhaust temperature to rise. If the temperature becomes higher than the maximum rated exhaust temperature, the temperature controller output will be lower than load speed controller output. Therefore, the minimum selector (whose inputs are received from temperature

and load speed controllers) will select the output of temperature controller and send it to fuel valve to reduce the exhaust temperature.

In Fig. 8.16, the difference between exhaust temperature and TC set point (maximum allowed temperature, i.e. 550 °C) is sent to a PI controller and then passes through the minimum selector going towards the servo valve controller.

- **Servo Valve Controller:** The fuel valve can control fuel flow injected to the gas turbine and causes mechanical torque to change according to operation logics. The input signal of servo valve comes from minimum selector gate, and the output is fuel valve aperture.

Figure 8.17 shows the servo valve controller in DIGSILENT. The $1/(1+ST)$ function is used to simulate valve dynamics. Other blocks (K/S , $1/ST$ and limiters) are implemented to consider the position of valve and the related aperture restriction. The output signal (Hg) goes to the turbine block.

- **Turbine Controller:** The output signal from the servo valve is imported to the turbine. The turbine dynamics is implemented in DSL by the $(1+ST_b)/(1+ST_a)$ function. IGV positioner is imported from the IGV controller and passes through the $K/(1+ST)$ function to model IGV dynamics. To properly measure the

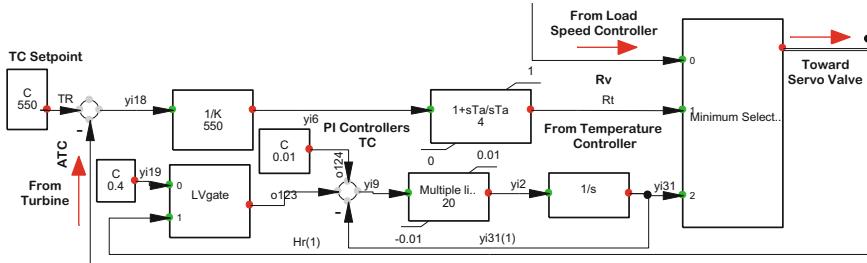


Fig. 8.16 Block definition of temperature controller implemented in DSL

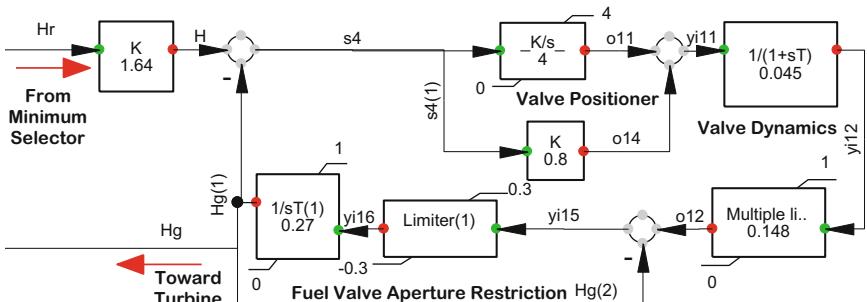


Fig. 8.17 Block definition of servo valve controller implemented in DSL

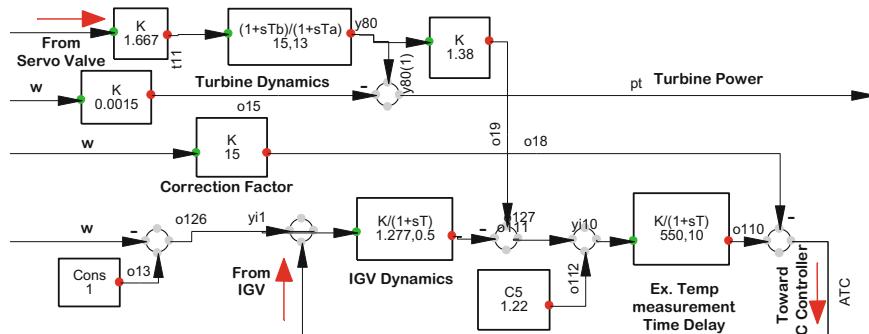


Fig. 8.18 Block definition of turbine model implemented in DSL

exhaust temperature, the related correction factor ($K = 15$) multiplied by the speed value (w) is used. The $K/(1+ST)$ function is used to simulate the temperature measurement time delay (Fig. 8.18).

All the state variables must satisfy $dx/dt = 0$ equation. Therefore, the turbine-governor initial conditions are as the following:

```

!The following equations are required for initial condition calculation
inc (x32)=1 !Temperature Controller, State Variable Initial Condition
inc (x33)=Hr+0.01 !Temperature Controller, State Variable Initial Condition
inc (SDR)=0 !Speed, Initial Condition
inc (y80)=pt+0.0015*w !Turbine Controller, Initial Condition
inc (x53)=y80 !Turbine Dynamics, State Variable Initial Condition
inc (Hg)=y80/1.667 !Valve Aperture, Initial Condition
inc (x27)=Hg !Valve Aperture Restriction, State Variable Initial Condition
inc (x26)=Hg/0.148 !Valve Dynamics, State Variable Initial Condition
inc (x25)=x26 !Valve Positioner, State Variable Initial Condition
inc (H)=Hg !Valve Input Signal, Initial Condition
inc (Hr)=H/1.64 !Minimum Selector Output Signal, Initial Condition
inc (Rv)=Hr !Load-Speed Controller Output Signal, Initial Condition
inc (x12)=0.3667*pgt !Generator Filter, State Variable Initial Condition
inc (x11)=0.4*o120 !Load-Speed Controller, State Variable Initial Condition
inc (x34)=0 !IGV Position, State Variable Initial Condition
inc (x29)=1.277*yi1 !IGV Dynamics, State Variable Initial Condition
inc (x30)=550*yi10 !Ex. Temp Controller, State Variable Initial Condition
inc (pset0)=0 !Active Power Setpoint in Change Mode, Initial Condition
inc (pset)=(Rv+0.4*0.3667*pgt)/(1.4*0.3667) !Active Power Setpoint, Initial Condition
inc (wref2)=1 !Frequency, Initial Condition
inc (change)=0 !Switch, Initial Condition

```

8.5.3 Scenario 1 (3GTs + Grid Connected + Import Power)

In this scenario, there are three generators, each one producing 80 MW, and the power plant is connected to the grid. The total active power of loads in service is 348 MW, so about 112 MW must be imported from the grid. As shown in

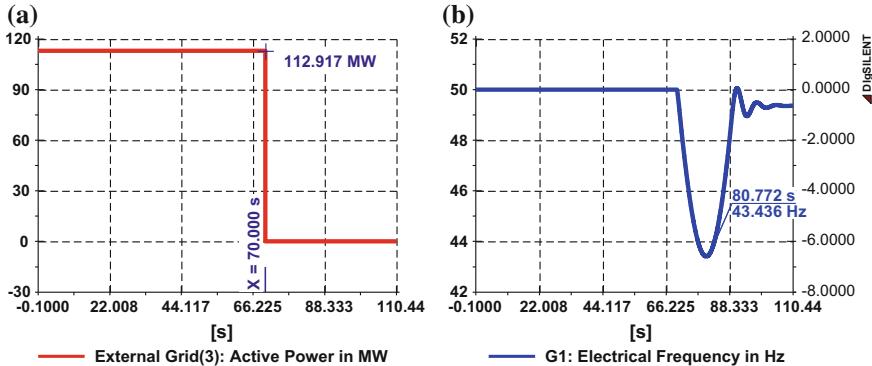


Fig. 8.19 Effect of plant disconnection without remedial action, **a** imported active power, **b** electrical frequency

Table 8.3 Description of scenario 1

Generation	3 GT (3 × 80 MW)—(2 GTs is in maintenance)
Load (MW)	About 348 MW
Import/export (MW)	Grid connected—import 112 MW (losses is considered)
Event	Islanding condition occurs (At $t = 70$ s, the plant was disconnected from the grid)
Effects of event	System frequency reaches 43.689 Hz Trip of generators.
Solution	PMS must shed 96 MW load at $t = 70.5$ s

Fig. 8.19, at $t = 70$ s, the plant is disconnected from the grid, and generation shortage causes system frequency to decrease to 43.436 Hz (the right axis shows frequency deviation from nominal value). Consequently, frequency relays send a trip command to generators which result in instability (Table 8.3).

Some loads must be shed to adjust the frequency to an acceptable margin in order to maintain stability and prevent economic loss. To achieve this aim, assuming that active power reserve of all generators is 5.5, 96 MW should be shed according to the Eq. (8.1) to regain frequency to predefined margin:

$$P_{\text{ToBeShed}} = +P_{\text{Import}} - (N_G \times P_{\text{Spin}}) + P_{\text{Offset}} = 112 - (3 \times 5.5) + 2.5 = 96 \text{ MW} \quad (8.7)$$

Therefore, according to Table 8.2, PMS intelligently selects and sheds four loads with priority 1 (i.e. Aromatics, Butanol, Benzene and Butenes with a total consumption of 97 MW) to maintain the stability of islanded plant. Figure 8.20 shows that shedding mentioned loads at $t = 70.5$ s causes the frequency to remain above 49.096 Hz. It must be noted that the total time from issuing the shedding command by PMS to perform it by lower level relays is considered to be 500 ms.

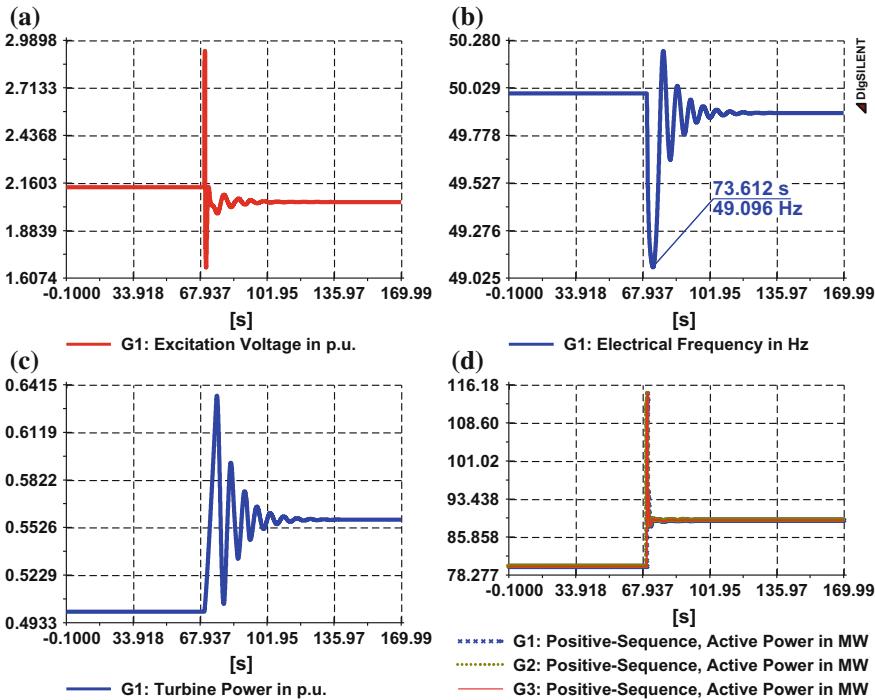


Fig. 8.20 Load shedding regains frequency to acceptable margin **a** excitation voltage **b** electrical frequency **c** turbine power **d** generator active power

As shown in Fig. 8.21 to increase turbine power, the fuel valve output must be increased. Consequently, the exhaust temperature will rise and IGV will open to be able to control temperature.

Timely execution of the remedial action is important, and fast reaction of PMS may be its main advantage over system operator. To show the importance of execution of the timely remedial action, above-mentioned load shedding procedure has been executed at various moments. Figure 8.22 shows that by increasing the load shedding execution time from 500 ms to 4 s, frequency reaches 45.996 Hz resulting in the operation of frequency relays and instability of the system.

Note In DIgSILENT, the outage of equipment is implemented by switch event. Figure 8.23 shows an outage of Tr230/400 which results in islanding mode.

8.5.4 Scenario 2 (3GTs + Island)

In addition to the fast response, another advantage of PMS over system operators is that PMS can select and execute optimum remedial actions in reasonable time to

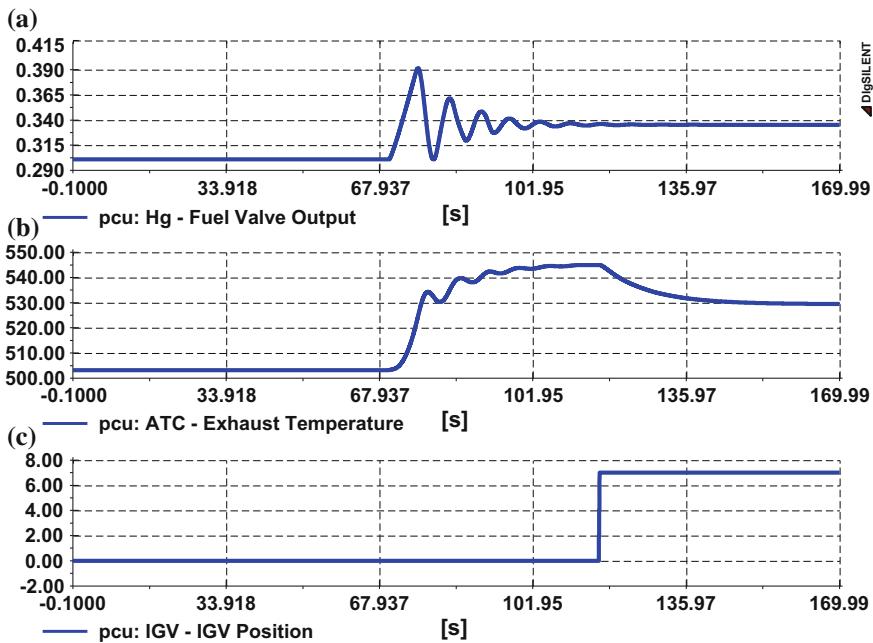
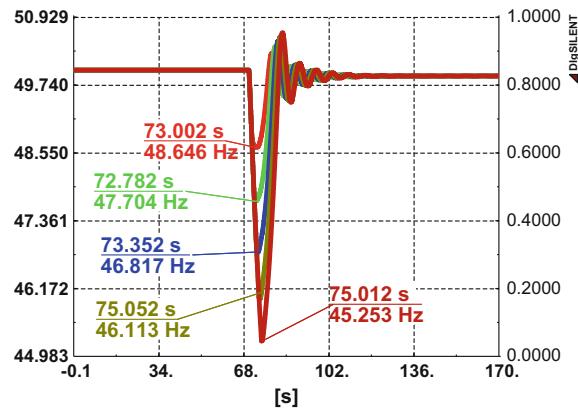


Fig. 8.21 aFuel valve output b exhaust temperature c IGV position

Fig. 8.22 Effects of the various execution time of load shedding



maintain system stability. For example, in condition mentioned in Table 8.4, three generators produce about 276 MW and the power plant is in islanding condition. In this case, if one of the GTs is tripped at $t = 70$ s, generation shortage will cause the frequency to decrease below an acceptable level (Fig. 8.24).

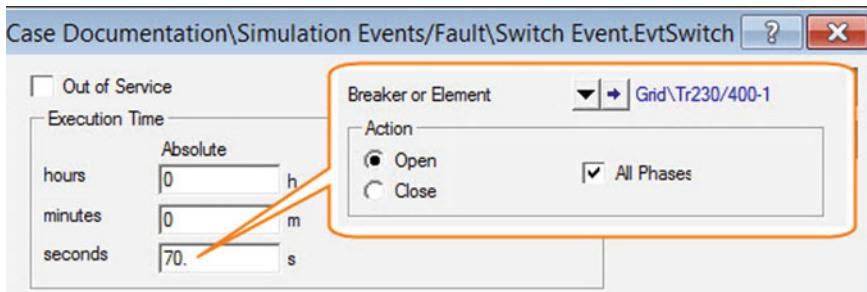


Fig. 8.23 Event related to an outage of Tr230/400-1 at $t = 70$ s

Table 8.4 Description of scenario 2

Generation	3 GT (3×92 MW)—(2 GTs is in maintenance)
Load (MW)	About 276
Import/export (MW)	Grid is in islanding condition (import/export 0 MW)
Event	Trip of one GT at $t = 70$ s
Effects of event	If the relay does not act, the frequency reaches 40.531 Trip of generators
Solution	83.5 MW load must be shed by PMS

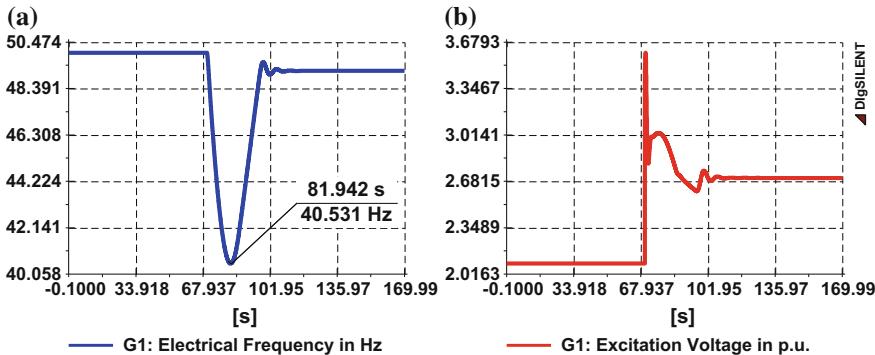


Fig. 8.24 Effect of trip without remedial action **a** frequency **b** voltage

To solve this problem, load shedding is inevitable. Suppose that because of power plant instruction, the operator cannot disconnect loads with priority 1. Thus, similar to the Eq. (8.1) and assuming that active power reserve of all generators is 5.5 MW, about 83.5 MW loads with priority 2 or higher level should be shed:

$$P_{\text{ToBeShed}} = P_{\text{Gen-Trip}} - (N_G \times P_{\text{Spin}}) + P_{\text{Offset}} = 92 - (2 \times 5.5) + 2.5 = 83.5 \text{ MW}$$

(8.8)

Hence, six steps of loads with priority 2 (i.e. Alachlore, Alpha-Olefins, Butadiene, Acid Acetic, Ethylene, and Gas Station) which totally consume 67 MW are shed. In this case, despite the execution of such remedial action, frequency reaches 48.126 Hz, and relay may act which results in system instability (Fig. 8.25a).

Whereas PMS is a smart system in a power plant, the command is released smartly from PMS so that the next step of the load is shed. In this case, the next step is Ethylhexanol. By adding a seventh step to load shedding program, about 97 MW load shedding is performed and minimum system frequency reaches a reasonable value (Fig. 8.25b).

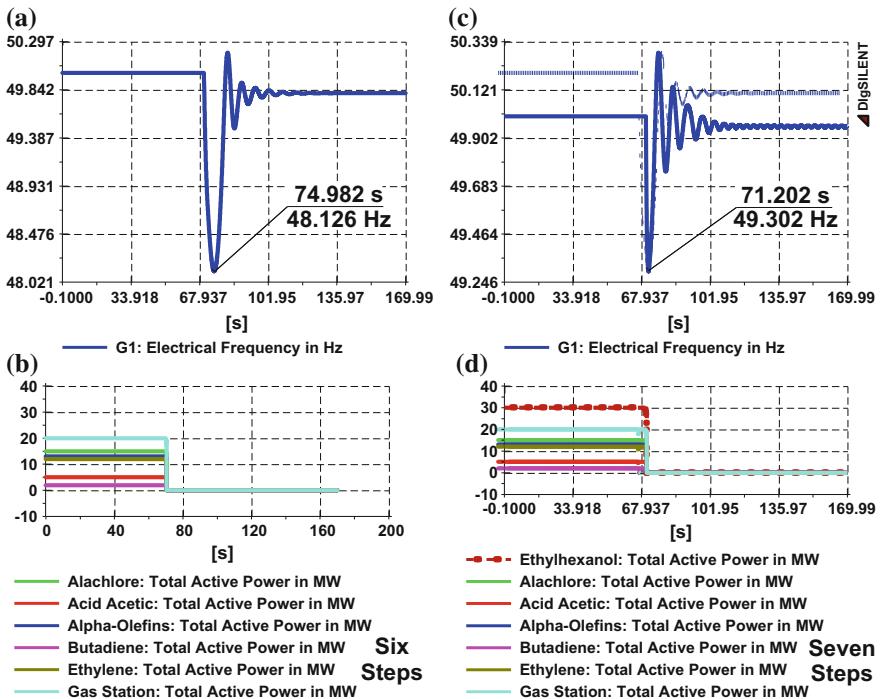


Fig. 8.25 Electrical frequency **a, b** after insufficient load shedding (6 steps), **c, d** after sufficient load shedding (7 loads)

8.5.5 Scenario 3 (3GTs + Gridconnected_Export_Generation Shedding)

In scenario 3, five generators produce 600 MW, the plant is connected to the grid, the load demand is 202 MW and about 398 MW is exported to the external grid. Figure 8.26 shows that if the power plant disconnects from the grid, the maximum and steady state values of frequency reach to 53.629 and 52.262 Hz, respectively. Excessive generation has to be shed to solve this problem. In this regard, PMS logic sheds one generator and changes governor mode of other generators to LOSC mode. As mentioned in Sect. 8.3.3, in LOSC mode, governor rapidly follows the change of frequency and tries to keep frequency in the reasonable region.

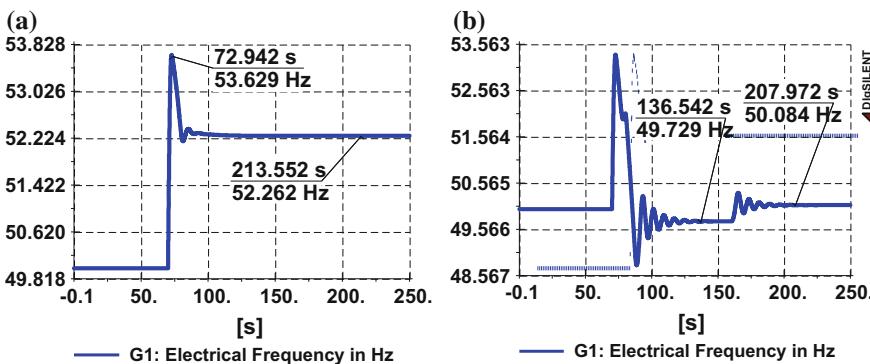


Fig. 8.26 Electrical frequency in scenario 3 **a** automatic change mode in governor logic does not act, **b** after execution of remedial actions

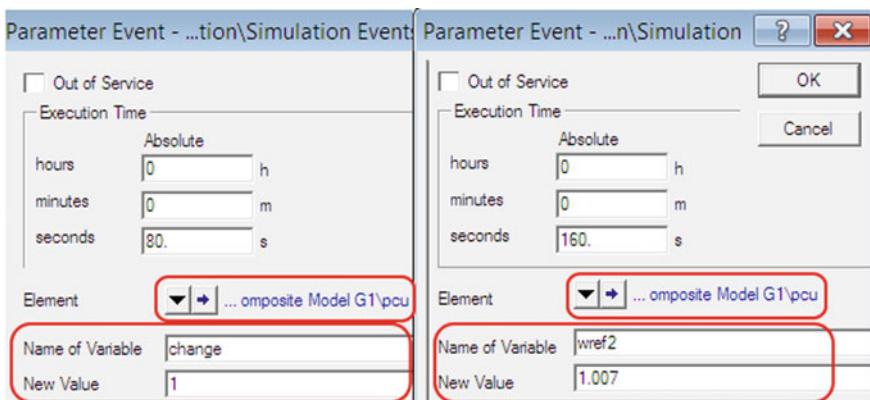


Fig. 8.27 Parameter event definition

Figure 8.26 shows the impact of the execution of such remedial action which causes the frequency to reach to 49.729 Hz in steady state condition.

Also, as mentioned in Sect. 8.3.5, secondary frequency control (SFC) can also be used to return system frequency in steady state condition to a nominal value by changing governor set point at about $t = 160$ s.

Note In DIgSILENT, parameter event (*.EvtParam) is used to change the parameter *change* in the turbine-governor model (Fig. 8.27) which alters turbine mode from LOLC to LOSC. Also, to change governor set point, parameter event is used to vary *wref2* in the turbine-governor model to adjust the frequency.

8.6 Conclusion

This chapter discusses PMS components, such as generation control (power and frequency), generation shedding, load shedding, I/E control and related PMS functions. It can control system frequency and voltage as well as generated real and reactive power by the shedding facility to prevent the cascade failure of the generation system. To execute this operation, according to the system configuration, PMS algorithm must calculate power imbalances, and wherever required, load/generation shedding is performed. Simulation of dynamic models in DIgSILENT simulation language (DSL) is required for the investigation of the transient stability. According to the accuracy of the implemented models, the results can be varied. Therefore, a proper implementation of controllers such as load/speed, temperature, IGV, servo valve, turbine and excitation systems is explained in detail and implemented in DigSILENT by a powerful tool (DSL). The power plant model considering related controllers with the changeover switch is used for functional testing of the PMS components. Different dynamic contingencies related to the islanded power system such as trip of grid connection are implemented. The results show that PMS plays a key role in the islanded power systems, and it operates during disturbances in order to preserve system stability.

References

1. H. Khoshkho, S. Shahrtash, Fast online dynamic voltage instability prediction and voltage stability classification. *IET Gener. Transm. Distrib.* **8**(5), 957–965 (2014)
2. T. Kato, H. Takahashi, K. Sasai et al., Multiagent system for priority-based load shedding in microgrid, in Proceedings of IEEE 37th Annual Computer Software and Applications Conference Workshops, (Kyoto, Japan, July 2013), pp. 540–545
3. U. Rudez, R. Mihalic, Predictive underfrequency load shedding scheme for islanded power systems with renewable generation. *Electr. Power Syst. Res.* **126** (2015)
4. GQ. Tang, Smart grid management & visualization, in *Emerging Technologies for a Smarter World (CEWIT), 8th International Conference, IEEE*, NY, 2011
5. K.E. Nicholson, R.L. Doughty, L. Mane et al., Cost effective power management systems. *Ind. Appl. Mag. IEEE* **6**(2), 23–33 (2000)

6. F. Pacheco et al., Power management system in an industrial plant, in *Petroleum and Chemical Industry Technical Conference (PCIC), 2012 Record of Conference Papers Industry Applications Society 59th Annual IEEE, Chicago, IL*
7. E. Roy Hamilton, J. Undrill et al., in *Considerations for generation in an islanded operation*, Copyright Material IEEE, Paper No. PCIC-2009-TBD
8. B. Cho, M. Almulla, H. Kim, N. Seeley, The application of a redundant load-shedding system for islanded power plants, in *proceedings of the 35th Annual Western Protective Relay Conference*, (Spokane, WA, October 2008)
9. Y.-Y. Hong, M.-C. Hsiao, Y.-R. Chang et al., Multiscenario underfrequency load shedding in a microgrid consisting of intermittent renewables. *IEEE Trans. Power Deliv.* **28**(3), 1610–1617 (2013)
10. D. Oliveira, A. Zambroni de Souza, A. Almeida et al., Microgrid management in emergency scenarios for smart electrical energy usage, in *Proceedings IEEE PowerTech*, (Eindhoven, Netherland, June 2015), pp. 1–6
11. KG. Ravikumar, T. Alghamdi, J. Bugshan, S. Manson, S. Krishna, Complete power management system for an industrial refinery, *IEEE Trans. Ind. Appl.* **52**(4), 2016
12. PMS description, circuit diagram and SIMATIC logics, power management grid stabilization and island operation, www.siemens.com
13. A. Parizad, Dynamic Stability Analysis for Damavand Power Plant Considering PMS Functions by DIGSILENT Software, 2013 13th International Conference on Environment and Electrical Engineering (EEEIC), 978-1-4799-2802-6. Nov 2013 IEEE
14. Dynamic Models Package Standard-1, “Tasks and data sheets, V 94.2, turbine-governor model”, Revision 1.7, SIEMENS, October 2012
15. SS. Iliescu et al., Gas turbine modelling for load-frequency control. *U.P.B. Sci. Bull. Series C* **70**(4), 2008
16. IEEE Standard Definitions for Excitation Systems for Synchronous Machines “Excitation system with static excitation and voltage feedback (IEEE-ST1)”, IEEE, July 2007

Chapter 9

Wide-Area Measurement, Monitoring and Control: PMU-Based Distributed Wide-Area Damping Control Design Based on Heuristic Optimisation Using DIgSILENT PowerFactory



Amin Mohammadpour Shotorbani, Sajad Madadi
and Behnam Mohammadi-Ivatloo

Abstract This chapter presents the design of a wide-area damping control (WADC) using a power system stabiliser (PSS) and remote PMU data from the wide-area measurement system (WAMS). The WAMS and monitoring architectures are described, and the common linear design techniques for proposing the WADC are introduced. An offline heuristic optimisation method is used to tune the WADC-PSS control parameters based on modal analysis. Generators' speed deviation is set as non-local inputs to PSS of the generators for WADC. PMU-based WADC improves damping of both the inter-area and the local oscillatory modes. A four-machine two-area power system is selected for numerical simulation using DIgSILENT PowerFactory. Modal analysis and time-domain simulation confirm the improved performance of the proposed WADC.

Keywords Distributed control · DIgSILENT PowerFactory · Smart grid Stability analysis · WADC · WAMS

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_9) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

A. M. Shotorbani (✉) · S. Madadi · B. Mohammadi-Ivatloo
Faculty of Electrical and Computer Engineering, University of Tabriz,
Tabriz, Iran
e-mail: a.m.shotorbani@tabrizu.ac.ir

S. Madadi
e-mail: sajad.madadi@gmail.com

B. Mohammadi-Ivatloo
e-mail: bmohammadi@tabrizu.ac.ir

9.1 Introduction

The operation and protection of the power system have been very intricate due to deregulation, power market activities, increased energy demand, the rapid load growth, expansion and interconnection of the grid and burgeoning penetration of the renewable energy resources. Consequently, the frequent operation of the power system near its operating limits [1] and recent major blackouts demonstrated the necessity of an advanced monitoring system and use of control methodologies.

Compared to the fast dynamics and transients of the power system, the conventional measurements through the metres, relays, transducers and the remote terminal units (RTUs) are very slow. Moreover, the conventional monitoring systems use local clocks as time stamps for the measured data. Therefore, exact phase angles calculation and dynamic monitoring, which are crucial for system stability, are challenging due to imprecise time tags. This deficiency leads to the invisibility of the very fast phenomena, insufficient control and inaccurate decisions [2].

This chapter will introduce the use of data from wide-area measurement system (WAMS) to improve the stability of the large-scale power system. The WAMS enhances the performance of the local power system stabiliser (PSS) and improves the damping of the system. The resultant wide-area damping control (WADC) provides advanced performance compared to local PSS if it is tuned properly. Consequently, the regulating techniques are introduced, and a model-free regulating scheme based on heuristic optimisation and modal analysis is proposed using PowerFactory. In the example project, WADC-PSS in a four-machine two-area power system is implemented. The control parameters are regulated using particle swarm optimisation (PSO) technique. Modal analysis is used to evaluate the optimality and to regulate the designed WADC iteratively.

Background: The wide-area measurement system (WAMS) was presented by Bonneville Power Administration (BPA) in the 1980s and was defined by Hauer in BPA as a strategic effort to meet critical information requirements of the changing power system [3]. The WAMS was used by the Department of Energy (DOE) and the Electric Power Research Institute (EPRI) in a project named as WesDINET in 1995 in order to provide dynamic information for the Western system since the Western System Coordinating Council (WSCC) recognised a variety of problems due to deficiency in available dynamic information [4]. The WAMS information was used to understand the arising disturbances, assess the current state of the system, avoid upcoming perturbations and preserve the reliability of the system during the deregulation and the restructuring of the power system.

In this regard, the advent of the phasor measurement units (PMUs) has introduced new functionalities to WAMS by providing synchrophasor measurements since 1994. Currently, WAMS uses the advantages of recent advances in the widespread use of PMUs and inexpensive high-speed communication technologies as well as the data from conventional measurement devices. WAMS is used in monitoring the states, enhancing the operation, control and protection of the large-scale interconnected power systems.

The information in WAMS is attained by processing the data in three subprocesses named as data acquisition by the measurement system, data transmitting through the communication infrastructure and data processing and analysis in the energy management system (EMS). The operational decisions are made based on the required information extracted from the received data and sent to remote controllers in addition to local actuators [5].

Necessity: Traditional state estimation in the power system is a nonlinear process. It uses the measured voltage, current, active and the reactive power flow and power injection to estimate the states of the system. Although state estimation using the SCADA data has acceptable practical accuracy during the steady-state and quasi-steady-state operation of the power system, the lack of accurate and synchronous measurements during the dynamic transients deteriorates the performance of the state estimation. Erroneous state estimation during the dynamic transient can also worsen the condition of the power system and may lead to instability [6]. Therefore, time-synchronised, high-rated and consistent measurements are required for monitoring the dynamics, to calculate the phase angles of the system phasors and to provide situational awareness by estimation of voltage stability margin [7].

In addition, the analysis of Northeastern US blackout in 2003 implies that preventive actions could have been made if the system's operators had the critical information about the falling states of the system and the indications of the upcoming instabilities. Advanced state monitoring could provide the opportunity to benefit the advantages of improved control methodologies and in-time decision-makings, recommended by the US–Canada Task Force on the Northeastern blackout.

9.2 Data Resources for the WAMS

The main advantages of the PMU-based WAMS compared to SCADA are an accurate and direct measurement of the phase angles with higher rates. It provides situational awareness and is suitable for monitoring the dynamics of the grid. Data, as the fundamental requirement of the functions in WAMS, have various sample rate, format, volume and different levels of importance. Data are classified into two categories as:

- The operational data including the phasor measurements and device status in the system.
- The logging data including the records of the events.

The continuous stream of the operational data is transmitted online to the control centre, whereas the logging data are transmitted offline in pre-specified occasions or when required. The operational data resources are the supervisory control and data acquisition (SCADA) and the synchronised phasor measurement system (SPMS). Some of the logging data resources are digital fault recorder (DFR), circuit breaker

monitor (CBM) and digital protective relay (DPR), to name but a few. A concise description of SCADA and SPMS as the operational data resources is presented in the following.

SCADA: In power transmission and distribution systems, SCADA is mainly used to gather the data from the system, monitor the state of the system and apply the control commands through long distances. Collecting the data, regulatory system control and alarm display is main functions of the SCADA system. The physical parts of SCADA system, including the master terminal units (MTUs), the remote terminal units (RTUs), programmable logic controllers (PLCs) and intelligent electronic devices (IEDs), are connected through communication protocols. Figure 9.1 illustrates the layout of the subsystem of SCADA [8]. In the following, functions of the subsystems of the SCADA (i.e. MTU and RTU) are briefly described.

The MTU is the centre for collecting and redirecting the data from RTUs, storing and analysing the received information and commanding the remote RTUs.

The RTU is the unit for monitoring data acquisition and controlling the remote equipment. Although the conventional RTUs communicate only with the MTU, modern RTUs are capable of communicating with other RTUs in a distributed architecture and may relay the data from other RTUs to MTU. In a hierarchical point of view, MTUs and RTUs act as masters and slaves, respectively [8].

SPMS: SPMS comprises phasor measurement units (PMUs), phasor data concentrator (PDC) and the communication systems. SPMS uses the time tags from the global positioning system (GPS) to stamp the measurements of voltage and current phasor values. Local frequency, the rate of change of frequency, negative and zero sequence components and the harmonics may also be measured through SPMS.

Fig. 9.1 Centralised SCADA architecture including the main subsystems

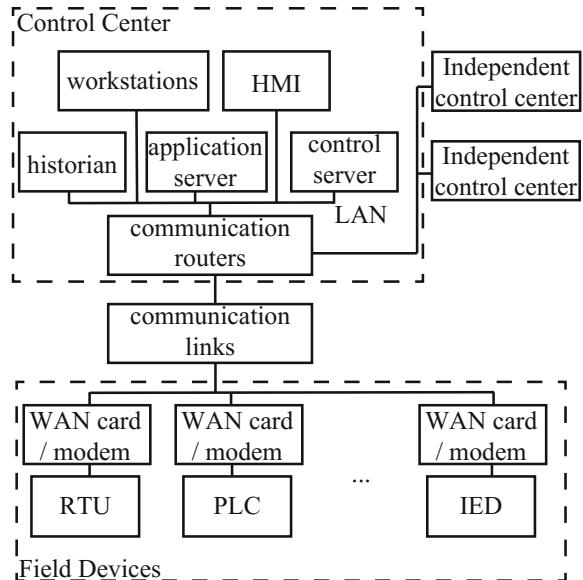
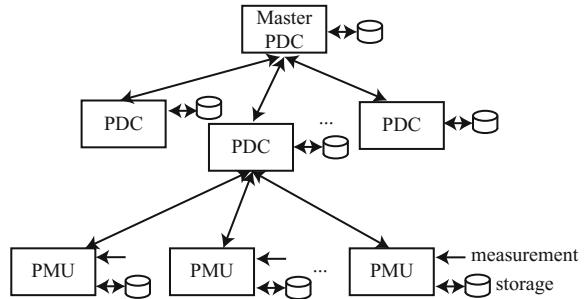


Fig. 9.2 SMPS architecture comprising the PDCs and the PMUs



Based on the time stamps, a detailed visualisation of the area is illustrated using the records of synchronised data [9]. PMUs and PDCs in SPMS have a similar role to RTUs and MTUs in SCADA, respectively. The hierarchical structure of a typical SPMS is depicted in Fig. 9.2 [10]. The functions of the subsystems of SPMS (i.e. PMU and PDC) are briefly described in the following.

The PDCs collect the data measured by the PMUs. The data are processed, the bad data are rejected, and the duplicate data are discarded. Subsequently, the data from the PMUs and other PDCs are aligned based on the time stamps. The reporting rate of data may vary for various PMUs. PDCs uniform the data with different rates and transmit the data to other PDCs, control centres or a super-PDC after alignment. Data transmission is conducted based on communication protocols. Nowadays, Internet protocols such as TCP, UDP and TCP/UDP are becoming popular for WAMS communications.

The PMU measures the AC voltage and current waveforms at a typical sampling rate up to 60 samples/cycle for a high degree of resolution. For synchronous measurements through the entire power system, the time stamps from the GPS are used by an oscillator leading to synchronisation accuracy of 1 μ s in measurements. Analogue signals are passed through an anti-aliasing filter and are digitalised by an A/D converter for processing. The phasor values and the local frequency are calculated using a digital signal processing system (DSP) [10]. Surge-suppressing filters are also used to filter the negative effects of the switching operation. The phase delay caused by the anti-aliasing filter is compensated before sending out the synchrophasor. PMUs communicate with the PDC at the rate of 30–60 samples/second [11]. Figure 9.3 shows the model of a typical PMU.

9.3 Applications of Synchrophasor in Modern Power System

The main application of the synchrophasor was recording and analysing the system's disturbances [11]. During its key role in post-event analysis, new applications were developed based on the useful information provided by the PMUs such as

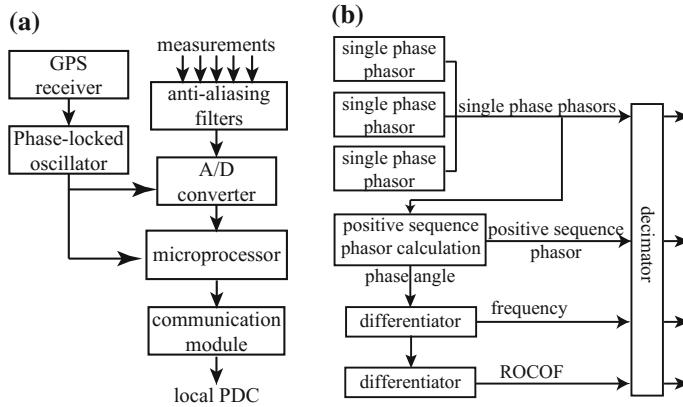


Fig. 9.3 **a** PMU block diagram and **b** PMU model

situational awareness and alarm management, state estimation, low-frequency oscillation damping, voltage stability and disturbance localisation.

Situational awareness is the power system's capability in predicting the future trajectories of state variables, detecting the potential events and notifying the evolving situations in the grid. Direct monitoring of the system's voltage/current phasors using the synchrophasor visualises the system state and provides accurate criteria for the stability assessment by monitoring the measured angles. The frequency monitoring network [2] and the real-time dynamics monitoring system [5] are two pilot examples of the early warning systems. The PMUs provide more accurate state estimation compared to the SCADA measurements, through direct measurement of the phase angles [12].

Progressive interconnection of large power systems over long distances often causes low-frequency oscillations (LFOs) [13]. Poorly damped inter-area oscillations may lead to system instability in addition to operational limitations. Also, in-time detection of the unstable oscillatory modes of the system could avoid unexpected outages, blackouts and widespread contingencies. Detection of the modes of the LFOs using the synchrophasor data is more convenient and has higher accuracy compared to SCADA measurements [14, 15].

Nowadays, the rapid load growth and high penetration of renewable energy resources have increased the risk of voltage instability. Traditional offline algorithms for voltage stability assessment require dynamic models of the power system. Despite conventional assessment methods, synchrophasor-based methods may use adaptive online modelling techniques, have less computation effort and provide higher accuracy in calculating voltage stability indexes [16].

9.4 Wide-Area Monitoring Architectures

The architectures of the wide-area monitoring system are classified into three main groups similar to basic control architectures as centralised, decentralised and distributed. These architectures differ in some aspects as the location of the decision-making and the actuating agents, and the flow of information between the agents.

In a centralised architecture, the data from all PMUs in the system are collected by a central PDC. The data are analysed, and the control decisions are made at the central PDC and are sent to control devices.

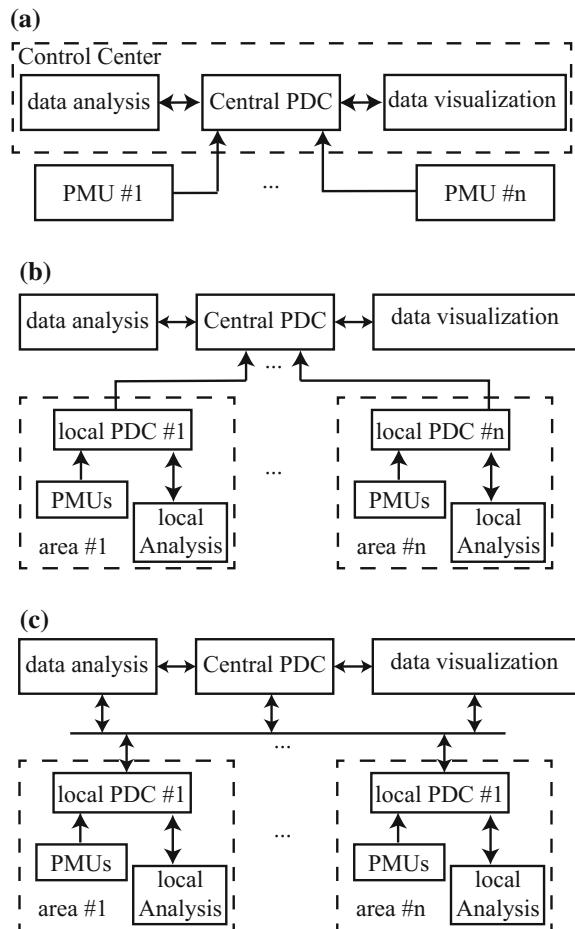
In a decentralised architecture, the area is considered as joined small subareas. Each subarea has its PDC and local controllers. The problem is divided into feasible subproblems, i.e. the local data analysis, local control commands and protective actions. The subproblems are mainly solved by the local control centres, i.e. local PDCs. However, communications with central controllers are made if the problem is not solvable with only local data. In a hierarchical architecture, the local PDCs are continuously connected with higher level PDCs through a low-bandwidth communication medium. Decentralised architecture has numerous advantages compared to centralised structure, such as the requirement of lower communication bandwidth, simple deployment, fast local control actions, robustness against communication failures, simple sharing of data, lower data storage requirements, higher data security and resiliency against the failure of decision centres. However, centralised architectures have the advantages of globally optimal performance, enhanced state monitoring, fast and precise management of the alarms and the alerts, and coordinated control and monitoring approach.

A distributed architecture involves master and supervisor PDCs at wide-area control level in addition to local PDCs at the region level. Each local PDC receives the data from the PMUs at its particular region. Despite decentralised architecture, the local PDCs are connected with each other as well as with some central PDCs for higher level coordination. In a distributed architecture, we benefit the advantages of both the centralised and decentralised architectures. This also avoids the deficiencies of the mentioned architectures. For maximum enhancement of the wide-area usage of the available network of PMUs, distributed architecture is preferred compared to the decentralised monitoring and control architecture in WAMS. Figure 9.4 illustrates the information flow in a centralised, decentralised and distributed wide-area monitoring and control architecture.

In a centralised architecture, one direct communication link is set between each PMU and the central PDC, whereas in a distributed architecture, a single data stream is set between the PMUs and the PDCs. Therefore, the cost of the communication links is also significantly reduced by using a distributed architecture.

Moreover, PMU data availability at the local PDCs (i.e. in a distributed architecture) may provide some additional functions as digital fault recording (DFR), power quality monitoring (PQM), relay setting and coordination among various strategies and also among different controlling devices.

Fig. 9.4 Communication architectures for wide-area measurement, monitoring and control: **a** centralised, **b** decentralised and **c** distributed



9.5 Wide-Area PSS Design for Damping Inter-area Oscillation

In this subsection, the basic principle and function of the conventional PSS and the WAMS-based PSS for oscillation damping is described. Furthermore, the recent linear design techniques used in WADC are reviewed. Based on a heuristic optimisation method, a WADC is designed in the next section for numerical simulation using DIgSILENT PowerFactory.

PSS is a supplementary control applied to the excitation field of the generator for improving the damping characteristic of the machine. The PSS modulates the field voltage to suppress the electromechanical oscillations in the speed and consequently in the output power through the automatic voltage regulator (AVR). Rotor angle oscillation may include low-frequency inter-area modes (in the range of

0.1–1.0 Hz), local modes (in the range of 1–2 Hz) and the modes of the plant (in the range of 2–3 Hz). WADC-PSS is an effective approach to damp inter-area and low-frequency oscillation [13].

The main approach in PSS design is applying the required electrical torque for damping the electromechanical oscillations. The PSS compensates the phase mismatch between the exciting system and the generator's electrical torque through lead compensators. The PSS input signal can be either generator speed deviation, output active power oscillation, the frequency deviation, or their combination. The main difference between using the frequency and the rotor speed as the PSS input signal is their impact on the sensitivity of the generator's terminal frequency to rotor oscillations. The frequency is more sensitive to inter-area modes compared to the local modes. Therefore, employing the frequency as the PSS input signal provides more effective inter-area modes damping rather than using the generator speed deviation as the PSS input signal. For an effective damping in a wide spectrum of oscillating modes, multi-band PSS schemes are used. Figure 9.5 shows the basic common structure of the linear PSS [17] and the proposed WADC-PSS scheme [13–17]. The residue method is one of the basic methods to design WADC system using the linear control techniques [18]. This method is also used in identification of the dominant eigenvalue [19] and modal decomposition WADC design [20].

The WADC is designed based on the PSS supplementary control and the data from the WAMS for simultaneous damping of the local and inter-area oscillation modes. The proposed WADC scheme based on the remote data using the WAMS is illustrated in Fig. 9.6. Generators' speed variations are set as the PSS non-local input signal. For damping a specific oscillatory mode, the weights $k_1, k_2 \dots k_n$ can be tuned in accordance with the participation factors of the input signals. Since we aim to increase the total stability of the system, all oscillatory modes are considered

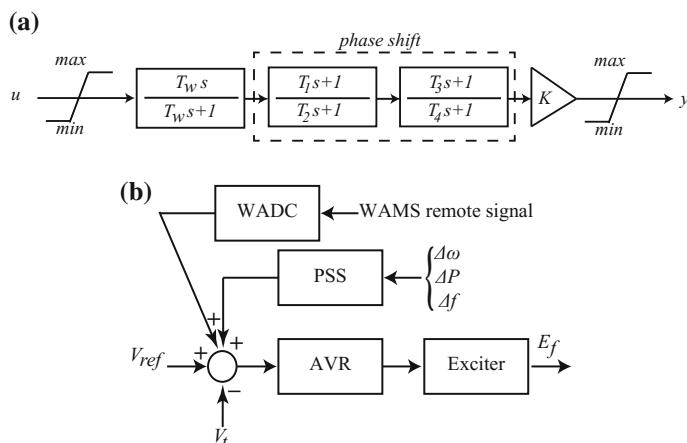
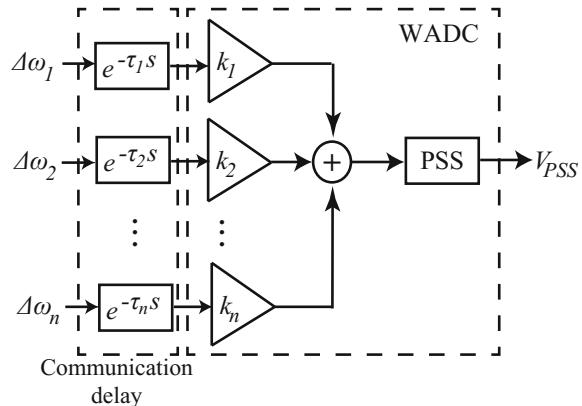


Fig. 9.5 **a** Basic PSS structure and **b** WAMS-based PSS (WADC)

Fig. 9.6 PSS structure with remote data input signal as a WADC



for tuning the weighting coefficients $k_1, k_2 \dots k_n$. The parameters of the proposed WADC-PSS are tuned based on the PSO algorithm, which is discussed in the next subsection. A basic approach to model the communication delay of the remote PMU signals is by assuming a constant delay in the control input argument of WADC [18], as depicted in Fig. 9.6. For frequency-domain implementation, the delay can be approximated using the Padé approximation.

9.6 Tuning the WADC-PSS Parameters with PSO Algorithm

Heuristic optimisation techniques can be used for offline parameter design of the WADC. In this technique, an objective function consisting real parts of all the eigenvalues of the system is considered for improving the damping through WADC. The control effort and the system state error can also be included in the objective function. Then, a heuristic iteration-based optimisation such as PSO [21, 22] is used for solving the optimisation problem.

In the following section, the PSO is used for tuning the wide-area PSS parameters in the numerical simulation of WADC. Consequently, the PSO algorithm is briefly described. PSO is a population-based optimisation algorithm. The particles in PSO are grouped into a swarm. Each particle makes use of the best position encountered by itself and that of its neighbours to proceed towards the optimal solution. The performance of each particle is evaluated using a predefined objective function (to be determined later). First, the algorithm parameters such as a number of particles, initial particles and velocities, and optimisation model parameters are initialised. Then, the algorithm starts with the initial swarm as initial solutions. If one of the termination conditions is satisfied, then the algorithm stops. Otherwise, the proposed procedure is iterated computing new velocities and positions as:

$$\forall i : \begin{cases} v_{k+1} = w_k v_k + r_1 c_1 (x_L - x_k) + r_2 c_2 (x_G - x_k) \\ x_{k+1} = x_k + v_{k+1} \end{cases} \quad (9.1)$$

where i , k , v , x , x_L and x_G are the number of particle, iteration, velocity, position, best local position of each particle, best global position of all particles, respectively. Also, r_1 and r_2 are random real numbers between 0 and 1, c_1 and c_2 are real constant positive numbers, and w is the inertia weight which is commonly chosen as an iteration-dependent decreasing function as:

$$w_{k+1} = \bar{w} - (\bar{w} - \underline{w})k/\bar{k} \quad (9.2)$$

where \bar{w} and \underline{w} are the selected maximum and minimum values of the inertia weight (e.g. $\bar{w} = 0.9$, $\underline{w} = 0.1$), and \bar{k} is the maximum iteration number.

For tuning the PSS and the WADC parameters, i.e. $\{k_1, k_2 \dots k_n, T_1 \dots T_4, K\}$, as in Figs. 9.5 and 9.6, we use the PSO algorithm. The objective function (9.3) is selected as the optimisation index for improving the damping performance of the PSS.

$$CF = \sum_{i=1}^m e^{\sigma_i} \quad (9.3)$$

where σ_i and m are the real part of the eigenvalue λ_i and the total number of the closed-loop poles, respectively.

Then, the constrained optimisation problem is formulated as:

$$\text{s.t. } \left\{ \begin{array}{l} \min CF \\ \zeta_i = -\frac{\sigma_i}{|\lambda_i|} \leq 0.05 \\ K_{\min} \leq K \leq K_{\max} \\ k_{\min} \leq k_i \leq k_{\max}, i = 1, 2, \dots, n \\ T_{i\min} \leq T_i \leq T_{i\max}, i = 1, 2, 3, 4 \end{array} \right. \quad (9.4)$$

The selected objective function (9.3) has large values for weakly damped modes. Therefore, the damping of the system modes will be increased by minimising (9.3). The first constraint in (9.4) assures the minimum 5% damping ratio for all modes.

The PSO algorithm is used to solve the constrained optimisation problem of (9.4).

9.7 Numerical Simulation

In this section, the performance of the proposed WADC-PSS is evaluated and compared to the local PSS scheme, using modal analysis. Parameters of the proposed WADC are selected through the PSO algorithm.

The test power system is modelled in DIgSILENT PowerFactory using the DIgSILENT simulation language (DSL), whereas the PSO algorithm and the modal analysis are implemented using DIgSILENT programming language (DPL).

A four-machine, two-area power system is used for simulating the proposed optimal WADC. Single-line diagram of the power system is depicted in Fig. 9.7. The machine and network parameters are selected as in [23]. All machines are equipped with PMU to construct a WAMS.

9.7.1 Modelling the Power System in DIgSILENT PowerFactory

PowerFactory provides an object-oriented environment for detailed simulation and a hierarchical scheme for modelling of a dynamic power system through standard models in addition to the user-defined DIgSILENT simulation language (DSL) models. Besides, the DIgSILENT programming language (DPL) provides an interface for task automation, iterative calculations and user-defined functions in the PowerFactory. Using DPL, one can define new scripts, variables, commands, new functions and mathematical formulae. DPL scripts are used through DPL command object (*ComDpl*). This object may also be used for configuration and preparation of the DPL scripts [24].

The power network is modelled in DIgSILENT PowerFactory as in Fig. 9.8, for simulation and modal analysis. Basic principles of modelling the power system components in DIgSILENT PowerFactory are completely covered in [24] and are not discussed here, for brevity.

The necessary components in implementing a WADC system in DIgSILENT PowerFactory consist of synchronous generators, automatic voltage regulators and

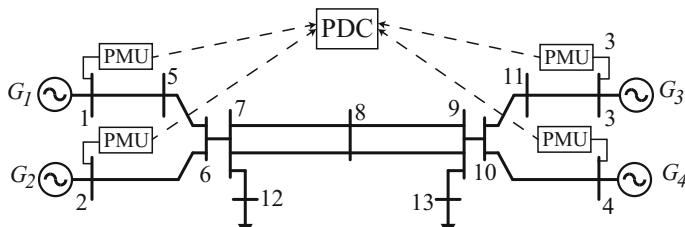


Fig. 9.7 Single-line diagram of a four-machine two-area power system [23]

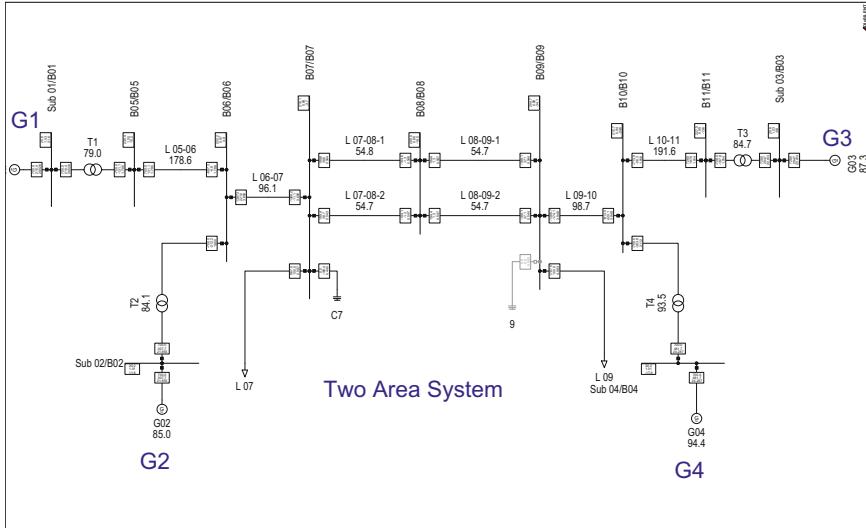


Fig. 9.8 PowerFactory implementation of the test power system

the PSS. The components are connected through a composite frame. In the composite model, each component is determined by a block. A schematic of the composite frame for the WADS is illustrated in Fig. 9.9, where $VPSS$, u_t and v_t are the output voltages of the PSS, the generator and the AVR, respectively.

Each component of the composite frame has an especial model named as a *Common Model*. Figure 9.10 shows a *Common Model* for the AVR structure. In this figure, input parameters are determined by voltage of the generator (u) and output voltage reference of PSS ($upss$). $usetp$ is the reference value which is determined by the user.

The common model of the proposed WADC is depicted in Fig. 9.11, where the angular velocities of the generators are classified as input parameters and are represented by w_i , $i=1,\dots,4$, and the generator's reference speed is shown by $wistep$, $i=1,\dots,4$.

9.7.2 Model Initialisation

Prior to simulation, initial values of the models' variables and parameters are set in accordance with the power flow calculations ($ComLfd$) in the pre-disturbance steady state of the system. The variables which are not included in the solution of the power flow calculation must be initialised manually, though the initial values of some variables are computed by the PowerFactory.

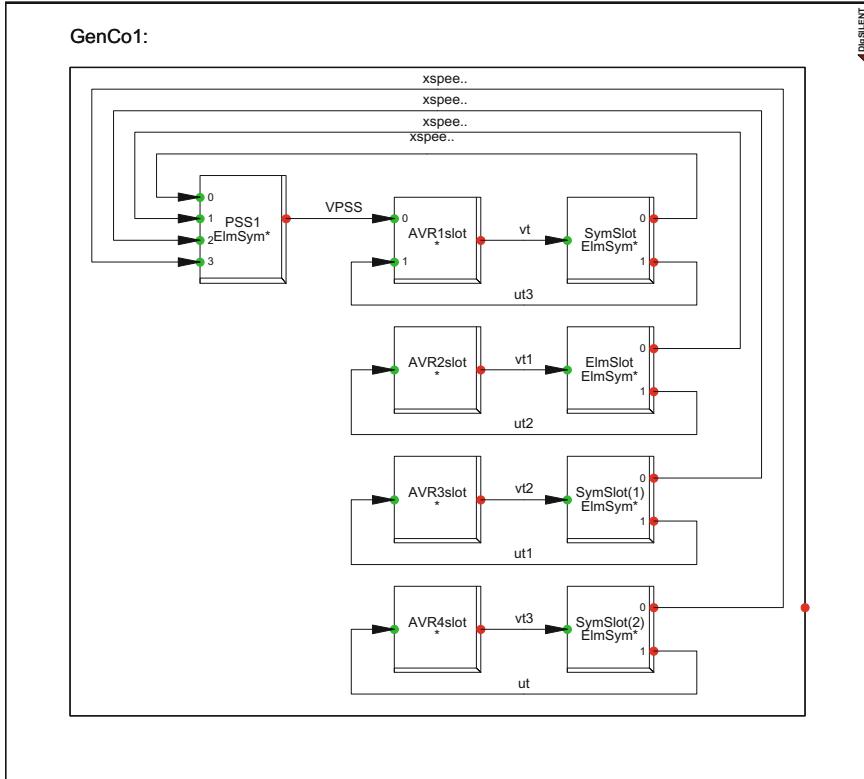


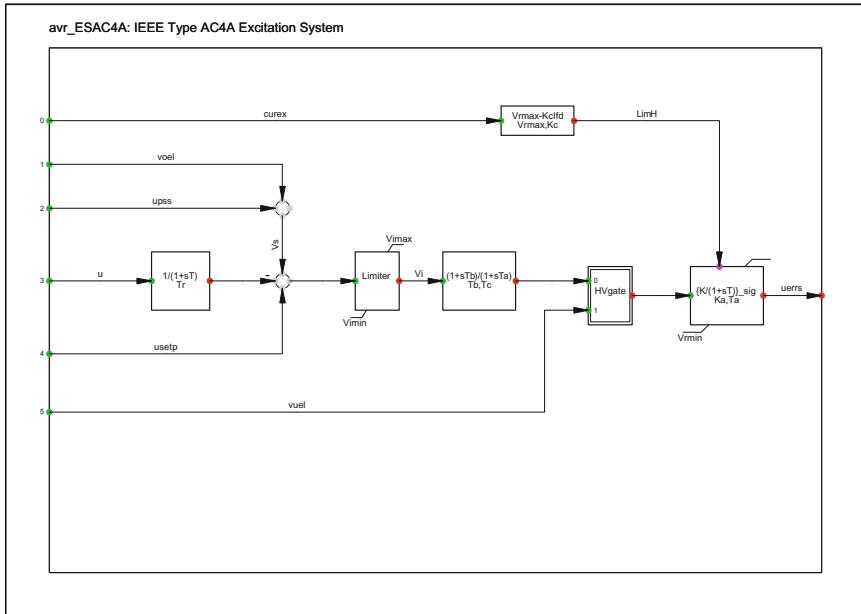
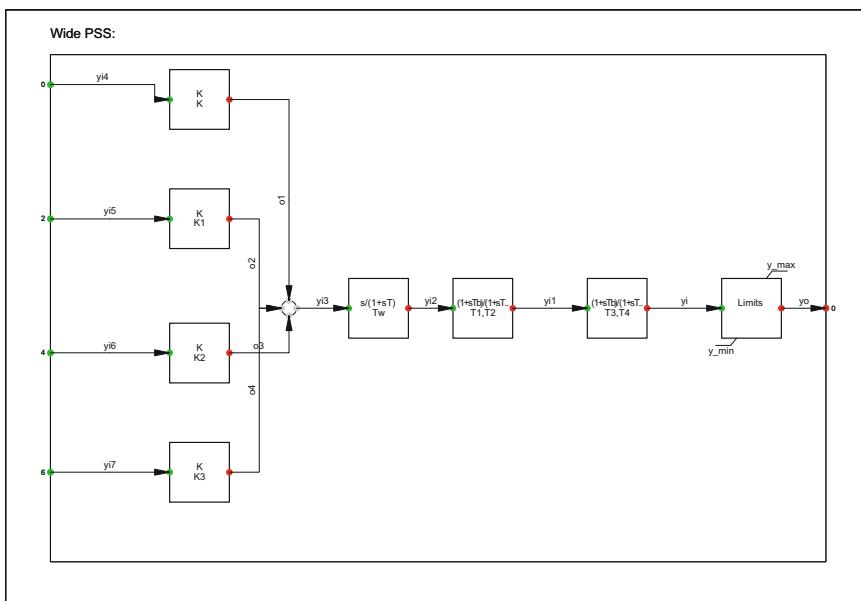
Fig. 9.9 Wide-area composite model in PowerFactory

In the hierarchical structure of DSL models, first the power system elements with the lowest hierarchy are initialised and then the process step up to the higher levels.

Figures 9.12 and 9.13 show initialisation of the AVR and the PSS using the additional equations, which is obtained through the power flow calculations (*ComLfd*) and considering that the system is in the steady state (i.e. $\dot{\mathbf{x}} = 0$).

9.7.3 Implementation of PSO Algorithm in PowerFactory

Implementation of the PSO algorithm in PowerFactory is discussed in this subsection. The first step for implementing the PSO is the definition of a DIgSILENT programming language (DPLCom) and some matrices (IntMat) and vectors (IntVac). Figure 9.14 illustrates the DPL implementation of the PSO algorithm. The objective function is determined by OF sub-DPL.

**Fig. 9.10** AVR PowerFactory model**Fig. 9.11** PSS PowerFactory common model

```
!### AVR Additional Equation ###
inc(xe)=ve
inc(xa)=Se+Ke*x
inc(xf)= xe
inc(xr)= ut
inc(userp)=ut+xa/Ka
inc(vf)=0
vardef(Tr)='s';
vardef(Ka)= 'pu';
vardef(Ta)='s';
vardef(Vrmax)='pu';
vardef(Vrmin)='pu';
vardef(Ke)= 'pu';
vardef(Te)='s';
vardef(Kf)= 'pu';
vardef(Tf)='s';
vardef(E1)= 'pu';
vardef(E2)= 'pu';
vardef(SE1)= 'pu';
vardef(SE2)= 'pu';
```

'Measurement Delay'
'Controller Gain'
'Controller Time Constant'
'Controller output limiter maximum value'
'Controller output limiter minimum value'
'Exciter Constant'
'Exciter Time Constant'
'Stabilization Path Gain'
'Stabilization Path Time Constant'
'Saturation Factor 1'
'Saturation Factor 3'
'Saturation Factor 2'
'Saturation Factor 4'

Fig. 9.12 AVR additional equation DPL

```
!### PSS Additional Equation ###
inc(x)=y14
inc(x1)=0
inc(x2)= 0
inc(upss)= 0
vardef(T1)='s';
vardef(T2)='s';
vardef(T3)='s';
vardef(T4)='s';
vardef(Tw)='s';
vardef(K)= 'pu';
vardef(K1)= 'pu';
vardef(K2)= 'pu';
vardef(K3)= 'pu';
```

'Time Constant'
'Time Constant'
'Time Constant'
'Time Constant'
'Time Constant'
'controler gain'
'controler gain'
'controler gain'
'controler gain'

Fig. 9.13 PSS additional equation DPL

The definition of the variables of the PSO algorithm is given in Fig. 9.15. The variables include the number of maximum iteration, the number of decision variables, the population size and the PSO algorithm parameters such as inertia weight, inertia weight damping ratio, personal learning coefficient and global learning coefficient.

The initialisation step of PSO algorithm is determined in Fig. 9.16. In this step, the value of variables is set to random values and saved into the position matrix, which is defined at the beginning of the initialisation code. Then, the PSS parameters are set by the initial values, and the objective function (9.4) is evaluated as shown in Fig. 9.17. The sub-DPL for evaluation of the objective function (i.e. OF.Execute()) is given in Fig. 9.23.

The local best and the global best are initially set based on the results of the initialisation step, as shown in Fig. 9.18. The local best is the best position of each

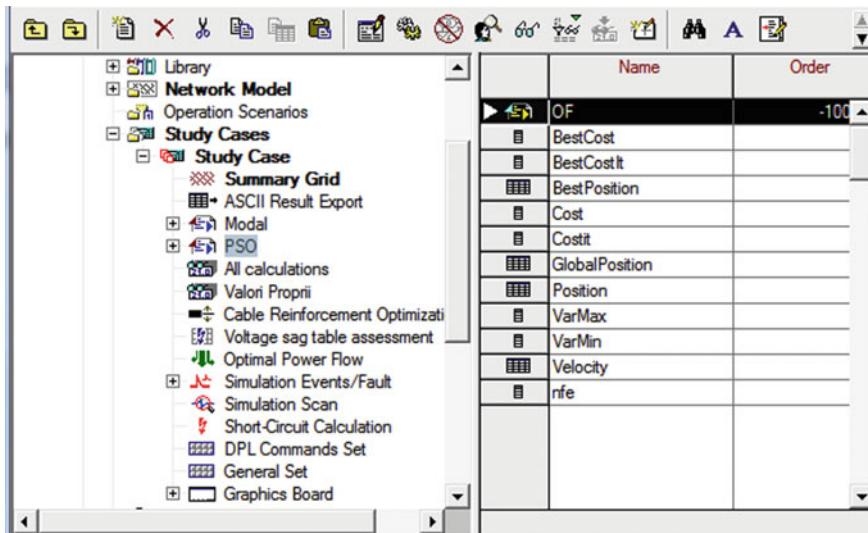


Fig. 9.14 PSO implementation DPL: vector definition page

```

! ### PSO Algorithm: variables definition #####
int nVar,Varsize,MaxIt,nPop,wdamp,i,j,index,it;
double phi1,phi2,phi,chi,w,c1,c2,aa,xx,yy,xxx,yyy,zz,zzz,GlobalBest,s;

ClearOutput();
EchoOff();
===== Problem Definition =====
nVar=9;                                !Number of Decision Variables
===== PSO Parameters =====
MaxIt=40;                                !Maximum Number of Iterations
nPop=20;                                  !Population Size (Swarm Size)
===== Constriction Coefficients =====
phi1=2.05;                                ! PSO Parameter
phi2=2.05;                                ! PSO Parameter
phi=phi1+phi2;                            ! PSO Parameter
chi=2/(phi-2+sqrt(phi*phi-4*phi));
w=chi;                                    !Inertia Weight
wdamp=1;                                  !Inertia Weight Damping Ratio
c1=chi*phi1;                            !Personal Learning Coefficient
c2=chi*phi2;                            !Global Learning Coefficient

```

Fig. 9.15 DPL implementation: PSO variables definition

particle in PSO algorithm, and the global best is the best position among all the particles. The best values are updated if a new optimum point is discovered.

After initialisation and updating steps, the main loop of the PSO algorithm is applied to find the best solution. This loop contains the steps of velocity update, position update, position limitation, objective function (OF) evaluation and local and global optimum update. The velocity update is depicted in Fig. 9.19.

The positions are updated using (9.1), and the velocity is computed in the previous step, as shown in Fig. 9.20. Subsequently, the positions are restricted to

```

!===== Initialization ======
Position.Init(nPop,nVar) ;           !Create best position Matrix for each population
BestPosition.Init(1,nVar) ;          !Create best position matrix for each variable
BestCost.Init(nPop);                !Create population cost matrix
Velocity.Init(nPop,nVar);           !Create the velocity limits matrix
GlobalBest=100000;                  !Initial value for best cost

for(i=1; i<=nPop; i=i+1) {
    for(j=1;j<=nVar;j=j+1){
        xx=Random();               !Position Initialization
        yy=VarMax.Get(j);           !set a random variable in [0-1]
        zz=xx*yy;                  !Get maximum value of jth variable
        Position.Set(i,j,zz);       !Generate a new population
        }                           !Save new population (zz) in the Position Matrix
}
}

```

Fig. 9.16 DPL implementation: PSO algorithm initialization

```

!===== OF Evaluation ======
for(i=1; i<=nPop; i=i+1) {
    aa=Position.Get(i,1);          !Get k1 (PSS Gain)
    OF:k1=aa;                     !Set k1
    aa=Position.Get(i,2);          !Get k2
    OF:k2=aa;                     !Set k2
    aa=Position.Get(i,3);          !Get k3
    OF:k3=aa;                     !Set k3
    aa=Position.Get(i,4);          !Get k4
    OF:k4=aa;                     !Set k4
    aa=Position.Get(i,5);          !Get t1 (PSS time constant)
    OF:t1=aa;                     !Set t1
    aa=Position.Get(i,6);          !Get t2
    OF:t2=aa;                     !Set t2
    aa=Position.Get(i,7);          !Get t3
    OF:t3=aa;                     !Set t3
    aa=Position.Get(i,8);          !Get t4
    OF:t4=aa;                     !Set t4
    aa=Position.Get(i,9);          !Get tw
    OF:tw=aa;                     !Set tw
    OF.Execute();                  !Evaluate the objective function
    aa=Cost.Get(1);                !Get the objective function value
}

```

Fig. 9.17 DPL implementation: OF evaluation step

the feasible area by limiting to the maximum and minimum limits, using the DPL codes given in Fig. 9.20. The limits are applied using *VarMax* and the *VarMin* vectors, which set the maximum and minimum values of variables, respectively.

The objective function (OF) is calculated after updating the positions (i.e. the variables). Figure 9.21 shows the OF evaluation using DPL. In the first stage, the updated positions are attained by reading the position matrix. Then, the OF is calculated using the “OF.Execute()” command.

Subsequently, the local and global best values are updated at each iteration, if new optimum values are discovered. The old values are replaced with new optimum positions. The DPL commands for the updating phase are described in Fig. 9.22.

```

!===== Initialize the Local Best =====
for(i=1; i<=nPop; i=i+1) {
    for(j=1;j<=nVar;j=j+1){
        xx=Position.Get(i,j);           !Get value of i population and j variable
        BestPosition.Set(i,j,xx);       !Set value of i and j variable in Best position
    }
    yy=Cost.Get(i);                 !Get cost calculated for ith population
    BestCost.Set(i,yy);             !Save cost of ith population in BestCost vector
}
!===== Initialize the Global Best =====
for(i=1; i<=nPop; i=i+1) {
    zzz= BestCost.Get(i);          !Get cost of ith population
    if (zzz<GlobalBest) {         !Compare cost of ith population with global best value
        GlobalBest=zzz;            !Set Global best value at cost of ith population
        index=i;                   !Save index of global best value
    }
}
for(j=1;j<=nVar;j=j+1) {
    xx=Position.Get(index,j);     !Get jth parameter of global best
    GlobalPosition.Set(1,j,xx);   !Save jth parameter of global best to GlobalPosition
}
BestCostIt.Init(MaxIt);          !Create BestCost vector of size MaxIt
nfe.Init(MaxIt);                !Create nfe vector of size MaxIt

```

Fig. 9.18 DPL implementation: PSO best value initialization step

```

!===== PSO main loop =====
for (it=1;it<=MaxIt;it=it+1){           !start of the PSO main loop
!===== Update Velocity =====
    for (i=1;i<=nPop;i=i+1){
        for (j=1;j<=nVar;j=j+1){
            xx=Velocity.Get(i,j);
            yy=Random();                  !create a random number
            zz=BestPosition.Get(1,j);
            xxx=Position.Get(i,j);
            zzz= Random();
            yyy= GlobalPosition.Get(1,j);
            aa=w*xx+c1*yy*(zz-xxx)+c2*zzz*(yyy-xxx);  !PSO velocity calculation
            Velocity.Set(I,j,aa);
        }
    }

```

Fig. 9.19 DPL implementation: PSO algorithm velocity update

The main loop (i.e. consisting Figs. 9.19–9.22) is iterated until the stopping criteria of the algorithm are met. Then, the optimum position values and the corresponding objective function are reported.

The objective function is written in the OF sub-DPL, which is connected to the main algorithm through the position and the cost matrices, as shown in Fig. 9.23. The PSS variables are defined in the input parameters window. The wide-area PSS is selected as an external object (i.e. the variable “a” corresponding to the generator 1) in Fig. 9.23, in order to change its parameters using the OF sub-DPL. The *cost* vector is defined to connect the sub-DPL with the PSO DPL. The eigenvalues are defined as *Eigen* and are stored in the *Valori Proprii* object.

```

=====
Position Update =====
for(i=1;i<=nPop;i=i+1){
  for (j=1;j<=nVar;j=j+1){
    xx=Velocity.Get(i,j); !Get velocity of ith population and jth variable
    xxx=Position.Get(i,j); !Get position of ith population and jth variable
    yy=xx+xxx;           !update the position
    Position.Set(i,j,yy); !Save new position
  }
}

=====
Position Limits =====
for(i=1;i<=nPop;i=i+1){
  for (j=1;j<=nVar;j=j+1){
    xx=Position.Get(i,j); !Get j-th position of the i-th variable
    zz=VarMax.Get(j);    !Get value of jth variable Maximum limit
    yy=VarMin.Get(j);    !Get value of jth variable Minimum limit
    if (xx<zz) {         !Compare the ijth position with maximum value
      Position.Set(i,j,zz);
    }
    if (xx>yy) {         !Compare the ijth position with minimum value
      Position.Set(i,j,yy);
    }
  }
}

```

Fig. 9.20 DPL implementation: PSO algorithm position update and limits

```

=====
OF Evaluation =====
for(i=1; i<=nPop; i=i+1) {
  aa=Position.Get(i,1);          !Get k1 (PSS Gain)
  OF:k1=aa;                     !Set k1
  aa=Position.Get(i,2);          !Get k2
  OF:k2=aa;                     !Set k2
  aa=Position.Get(i,3);          !Get k3
  OF:k3=aa;                     !Set k3
  aa=Position.Get(i,4);          !Get k4
  OF:k4=aa;                     !Set k4
  aa=Position.Get(i,5);          !Get t1 (PSS time constant)
  OF:t1=aa;                     !Set t1
  aa=Position.Get(i,6);          !Get t2
  OF:t2=aa;                     !Set t2
  aa=Position.Get(i,7);          !Get t3
  OF:t3=aa;                     !Set t3
  aa=Position.Get(i,8);          !Get t4
  OF:t4=aa;                     !Set t4
  aa=Position.Get(i,9);          !Get tw
  OF:tw=aa;                     !Set tw
  OF.Execute();                 !Evaluate the objective function
  aa=Cost.Get(1);               !Get the objective function value
  Costit.Set(i,aa);             !Set this value for iteration
}

```

Fig. 9.21 DPL implementation: OF evaluation step

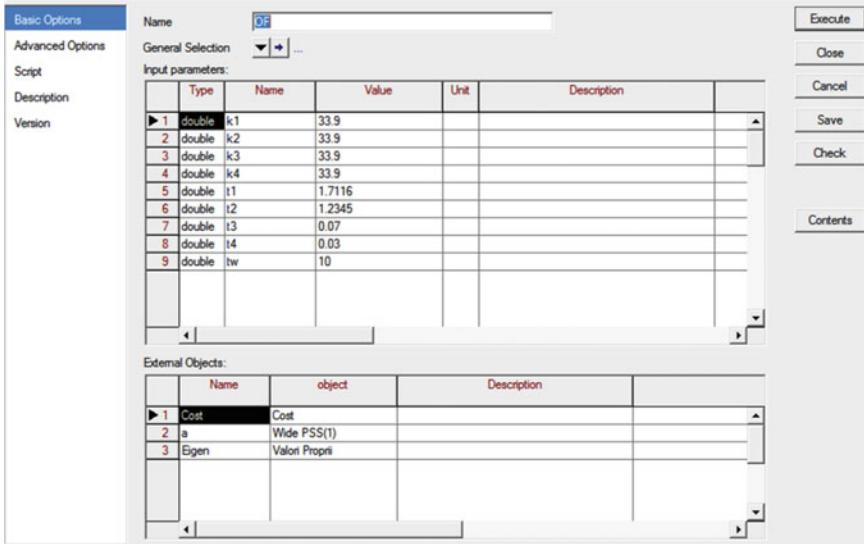
The parameters presented in value column of Fig. 9.23 are the initial values of the WADC gains and time constants which are altered using the PSO script, and the optimal parameters are achieved finally.

Figure 9.24 shows the first stage in implementation of the OF sub-DPL, where the required variables and commands are defined. The commands for computing the initial condition and the modal analysis are defined as objects and executed.

```

===== Update local Best =====
for(i=1; i<=nPop; i=i+1) {
    xx=Cost.Get(i);           !Get cost of ith population
    yy=BestCost.Get(i);       !Get best cost of ith population
    if (xx<yy) {
        BestCost.Set(i,xx);   !Compare the values
        !Save best cost
        for(j=1;j<=nVar;j=j+1){
            xx=Position.Get(i,j);
            BestPosition.Set(i,j,xx); !Get position of ith population and jth variable
            !Save best population
        }
    }
}
===== Update Global Best =====
for(i=1; i<=nPop; i=i+1) {
    zzz= BestCost.Get(i) ;      !Get best cost of ith population
    !Campare best cost of ith population with Global best value:
    if (zzz<GlobalBest) {
        GlobalBest=zzz;          !Save global value
        index=i;                 !Save local of global value
        for(j=1;j<=nVar;j=j+1) {
            xx=Position.Get(index,j); !Get jth variable of Global best
            GlobalPosition.Set(1,j,xx); !Save jth variable of Global best
        }
    }
}
BestCostIt.Set(it,GlobalBest); !Save the best Cost in it-th iteration
w=w*w damp;                  !update w
}
!Terminate the PSO main loop started at Fig. 19

```

Fig. 9.22 DPL implementation; updating global and local best values**Fig. 9.23** Structure of the OF sub-DPL for PSO objective function

```
!### Objective Function ###
!=====
object Inc,Modal,Eigenvalues,oVi;
int Nvar,Nval,a1,yy,ww,xx,xxx,www,Mo1,Mo2,zzz,ncol,nrow;
double real,imaginary,ss,Pg,pareal,paimag,Eig2,past,prob;
ClearOutput();
EchoOff();
Inc=GetCaseCommand('ComInc');
Modal=GetCaseCommand('Modal.ComMod');
Inc.Execute();                                !Definition of initial calculation
Modal.Execute();                             !Definition of Modal calculation
!Execute initial calculation
!Execute Modal calculation
```

Fig. 9.24 Definition codes of objective function

```
!=====
object Eigen;
nrow=Eigen.NRow();                           !Set number of rows
ncol=Eigen.NCol();                           !Set number of columns
LoadResData(Eigen);                         !Load the Eigenvalues
Nvar=ResNvars(Eigen);
Nval=ResNval(Eigen,0);
ResulModal.AddVars(this,'b:real','b:imaginary'); !Set real part of Eigenvalues
ResulModal.Init();                           !ResulModal creation
ss=Vec.Size();                             !Get size of vector
www=0;                                     !Set a zero matrix of size Nvalx2
S.Init(Nval,2,0);
```

Fig. 9.25 Set the Eigen matrix

In Fig. 9.25, the eigenvalues are calculated through modal analysis, and the results are saved into the “*Eigen*” object. Size of real and imaginary parts of the eigenvalues is set by “*NRow*” and “*NCol*” commands, to construct a matrix of size $Nval \times 2$, named as *S*.

In Fig. 9.26, the impact of the PSS parameters on the eigenvalues is evaluated by calculating the objective function (9.4). First, the cost variable is set to zero, and the modal analysis is executed to compute the eigenvalues. Then, the *Eigen* object is loaded, and the real part of the eigenvalues is extracted. The objective function (9.4) is calculated based on real part of the eigenvalues.

Using PSO, the WADC optimal parameters are designed as given in Table 9.1.

9.7.4 *Modal Analysis Using PowerFactory*

The modal analysis in PowerFactory is obtained either after reaching a balanced steady-state condition through a dynamic simulation or after an RMS simulation and is enabled once the initial condition is calculated. The default state variable for the modal analysis is the generators’ speed, which can be altered using the *advanced options* through the *settings* at the modal analysis *basic options* page [25].

After calculating the initial condition, a system matrix is constructed by modal analysis for the calculation of the eigenvectors following the model linearisation by

```

===== objective function Calculation Step =====
Inc.Execute();                                !Initialization
Modal.Execute();                               !Modal Analysis
LoadResData(Eigen);                          !Loading the eigenvalues
Nvar=ResNvars(Eigen);
Nval=ResNval(Eigen,θ);
a1=2;
yy=0;
ww=0;
cost=0;
for(xx=0;xx<Nval;xx=xx+1){
    GetResData(real,Eigen,xx,yy);           !extracting the real part of Eigen
    GetResData(imaginary,Eigen,xx,yy+1);    !extracting the imaginary part of Eigen
    ww=ww+1;
    pareal=S.Get(ww,a1-1);
    cost=cost+exp( pareal);                !Get real part of eigenvalues
    Cost.Set(ww,cost);                     !Evaluate the objective function
    ResulModal.WriteDraw();                 !Save objective function value
}                                              !Save the data to ResulModal
}

```

Fig. 9.26 DPL implementation: cost calculation step**Table 9.1** WADC-PSS parameters designed by the PSO algorithm

T_w	T_1	T_2	T_3	T_4	k	k_1	k_2	k_3	k_4
10	0.8076	0.5498	0.5180	0.4771	16.425	2.1093	1.7011	1.2964	1.3015

PowerFactory. The participation factors, the left, and the right eigenvectors can be calculated on the *advanced options* page.

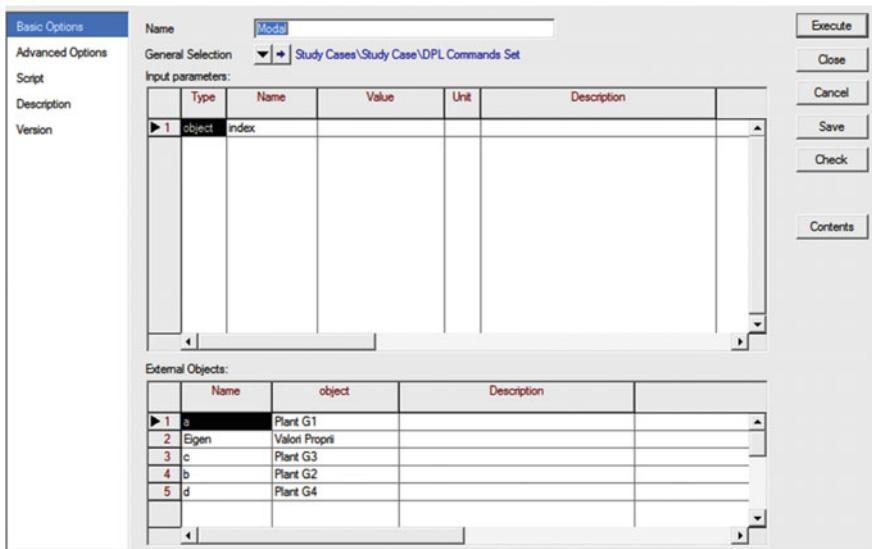
The results of the modal analysis are stored in the object named as *ElmRes* and can be outputted by the *ComSh* command. By selecting the *Eigenvalues* at the *LoadFlow/Simulation* part (from the *Output of Results* page), the *Output of Eigenvalues* part will be enabled. In this part, one can choose to report the results of the calculated eigenvalues or the participation factors at the output window. The participation results can further be filtered based on *Maximal Damping*, *Maximal Period* and *Min. Participation*.

Results of the modal analysis can be plotted using *VisModephasor*, *VisModbar* and the *Viseigen* plotting options available at the *Select Plot Type* menu. *VisModephasor* provides the phasor diagram of the participation factors. *VisModbar* shows the magnitudes of the generators' participation factors. *Viseigen* plots the eigenvalues in the real–imaginary coordinate system. The imaginary axis can also be altered to represent the frequency and period of the eigenvalues.

Structure of the modal analysis sub-DPL is shown in Fig. 9.27. Four local PSSs (i.e. a, b, c, d) for plants G1–G4 are defined in Fig. 9.27a, whereas a wide-area PSS (i.e. a) is defined for plant G1 in Fig. 9.27b. The PSS units for each plant are selected as an external object (i.e. a, b, c, d), in order to change its parameters using the PSO DPL.

The modal analysis DPL script for the simulation scenarios is classified into four parts including a definition part and three scenarios as: Scenario 1: the system without PSS, Scenario 2: local PSS at each plant and Scenario 3: the WADC with wide-area PSS at plant 1.

(a)



(b)

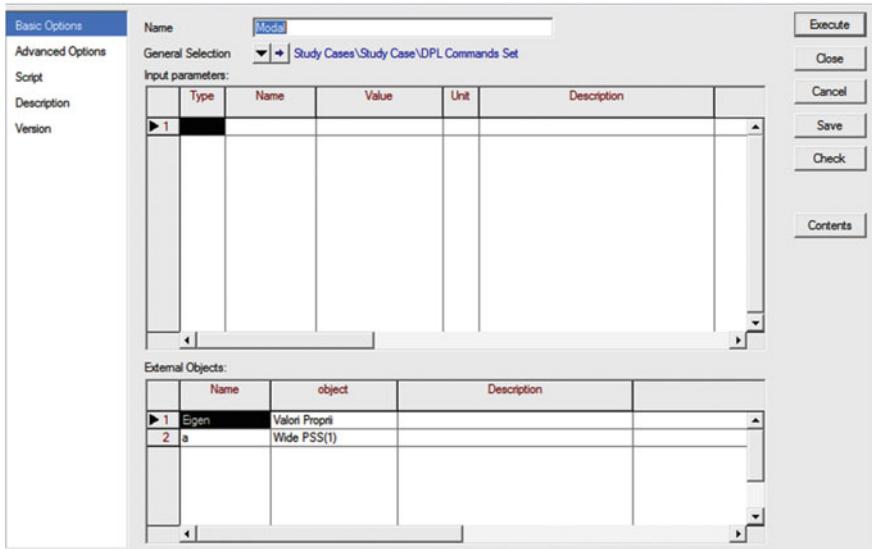


Fig. 9.27 Structure of the modal analysis DPL: **a** Local PSSs at each plant and **b** WADC-PSS

```

### Modal Analysis ###
object Inc,Modal,Eigenvalues,oVi;
int Nvar,Nval,a1,yy,ww,xx,xxx,www;
double real,imaginary,ss,Pg,pareal,paimag,Eig2,past;
ClearOutput();
EchoOff();
Inc=GetCaseCommand('ComInc');
Modal=GetCaseCommand('Modal.ComMod');
Inc.Execute();
Modal.Execute();
LoadResData(Eigen);
Nvar=ResNvars(Eigen);
Nval=ResNval(Eigen,0);
ResModal.AddVars(this,'b:real','b:imaginary'); !Extract real/imaginary parts
ResModal.Init(); !Create ResulModal
ss=Vec.Size(); !Determine size of vector

```

Fig. 9.28 DPL implementation: definition of the objects and the parameters of modal analysis

```

!===== Modal Analysis; System Without PSS =====
Eigen1.Init(Nval,2,0);
a:outserv=1; !out servicing the local PSS1
b:outserv=1; !out servicing the local PSS2
c:outserv=1; !out servicing the local PSS3
d:outserv=1; !out servicing the local PSS4
Inc.Execute(); !Execute initialization
Modal.Execute(); !Execute modal calculation
LoadResData(Eigen); !Load Eigen
Nvar=ResNvars(Eigen);
Nval=ResNval(Eigen,0);
a1=2;
yy=0;
ww=0;
for(xx=0;xx<Nval;xx=xx+1){
    GetResData(real,Eigen,xx,yy); !Get real part of xx-th eigenvalue
    GetResData(imaginary,Eigen,xx,yy+1); !Get imaginary part of xx-th eigenvalue
    yy=yy+1;
    Eigen1.Set(ww,a1-1,real); !Save real part of the eigenvalue
    Eigen1.Set(ww,a1,imaginary); !Save imaginary part of the eigenvalue
    ResulModal.WriteDraw(); !Save the data to ResulModal
}

```

Fig. 9.29 Modal analysis without PSS units

The DPL script for the definition part is given in Fig. 9.28, where the parameters and the objects required to execute the modal analysis are defined.

The first scenario is the modal analysis without the PSS units, which is achieved by out servicing the PSS units, as shown in Fig. 9.29. The command “*PSS-unit-name:outserv=1*” is used to out service the PSS where *PSS-unit-name* is the unit name as a, b, c and d. After out servicing the local PSS, the eigenvalues are calculated and decomposed into real and imaginary parts. Then, the eigenvalues are saved in *Eigen1*.

The second scenario is the modal analysis with local PSS units, which is achieved by activating the PSS units, as shown in Fig. 9.30. The command “*PSS-unit-name:outserv=0*” is used to activate the PSS where *PSS-unit-name* is the unit name as a, b, c and d. Then, the eigenvalues are calculated and decomposed into real and imaginary parts. The eigenvalues are saved in *Eigen2*.

```

!===== Modal Analysis; System With local PSS =====
a:outserv=0;                                !local PSS1 in service
b:outserv=0;                                !local PSS2 in service
c:outserv=0;                                !local PSS3 in service
d:outserv=0;                                !local PSS4 in service
Inc.Execute();                               !Execute initialization
Modal.Execute();                            !Execute modal calculation
LoadResData(Eigen);                         !Load eigenvalues
Nvar=ResNvars(Eigen);
Nval=ResNval(Eigen,0);
a1=2;
yy=0;
ww=0;
for(xx=0;xx<Nval;xx=xx+1){
  GetResData(real,Eigen,xx,yy);           !Get real part of xx-th eigenvalue
  GetResData(imaginary,Eigen,xx,yy+1);    !Get imaginary part of xx-th eigenvalue
  yy=yy+1;
  Eigen2.Set(ww,a1-1,real);              !Save real part of the eigenvalue
  Eigen2.Set(ww,a1,imaginary);            !Save imaginary part of the eigenvalue
  ResulModal.WriteDraw();
}

```

Fig. 9.30 Modal analysis with local PSS units for each plant

The third scenario is the modal analysis with local PSS units, which is achieved by activating the WADC-PSS unit at plant 1, as shown in Fig. 9.31. The local PSS is out serviced, and the WADC-PSS is activated by using the command “wapsst outserv=0”.

The optimal parameters of the WADC-PSS unit are calculated using the PSO algorithm, which have been stored in the *PSO* matrix. The modal analysis of the proposed WADC-PSS with optimal parameters is executed using the DPL script in Fig. 9.31. The eigenvalues are calculated and decomposed into real and imaginary parts. Then, the eigenvalues are saved in *Eigen3*.

Modal analysis of the uncompensated system shows that the inter-area modes 15 and 16 ($0.078 \pm j3.521$) are unstable modes. The inter-area modes in right eigenvector of the machines’ angles are depicted in Fig. 9.32.

Real part of the participation factor of the machines in the mentioned modes is shown in Fig. 9.33. Positive participation factors show that the increase of damping torque of any machine will improve the damping of the modes 15 and 16.

Furthermore, the modes 17 and 18 ($-0.64 \pm j6.82$) and the modes 19 and 20 ($-0.64 \pm j6.82$) have lower damping, compared to other modes. Therefore, we aim to stabilise the unstable modes and increase the damping of the weakly damped modes using WADC. The weighting coefficients for the third and the fourth machines are negative values.

For further comparison, the PSS parameters are also tuned based on the local measurements as presented in Table 9.2 [23].

Table 9.3 shows the modal analysis results using PowerFactory for the three scenarios: without PSS, with local PSS and the WADC-PSS. The unstable and the weakly damped modes (i.e. modes 15–20) are compared in Table 9.3, for the uncompensated power system and the system with the proposed WADC.

```

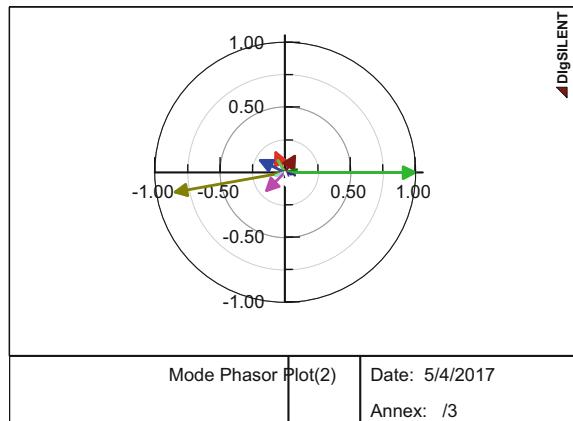
=====
 Modal Analysis; system with WADC PSS =====
aa= PSO.Get(1);                                !Get k1 in PSO vector
a:K=aa;
aa= PSO.Get(2);
a:K1=aa;
aa= PSO.Get(3);
a:K2=aa;
aa= PSO.Get(4);
a:K3=aa;
aa= PSO.Get(5);                                !Get T1
a:T1=aa;
aa= PSO.Get(6);                                !Set T1
a:T2=aa;
aa= PSO.Get(7);
a:T3=aa;
aa= PSO.Get(8);
a:T4=aa;
aa= PSO.Get(9);                                !Get Tw
a:Tw=aa;                                         !Set Tw

Inc.Execute();                                    !Execute initial calculation
Modal.Execute();                                 !Execute modal calculation
LoadResData(Eigen);                            !Load eigenvalues
Nvar=ResNvars(Eigen);
Nval=ResNval(Eigen,θ);
a1=2;
yy=0;
ww=0;
for(xx=0;xx<Nval;xx=xx+1){
  GetResData(real,Eigen,xx,yy);                !Get real part of xx-th eigenvalue
  GetResData(imaginary,Eigen,xx,yy+1);          !Get imaginary part of xx-th eigenvalue
  yy=yy+1;
  Eigen3.Set(ww,a1-1,real);                    !Save real part of the eigenvalue
  Eigen3.Set(ww,a1,imaginary);                  !Save imaginary part of the eigenvalue
ResultModal.WriteDraw();
}

```

Fig. 9.31 Modal analysis with WADC-PSS unit at plant 1

Fig. 9.32 Mode phasor plot of the participation factors



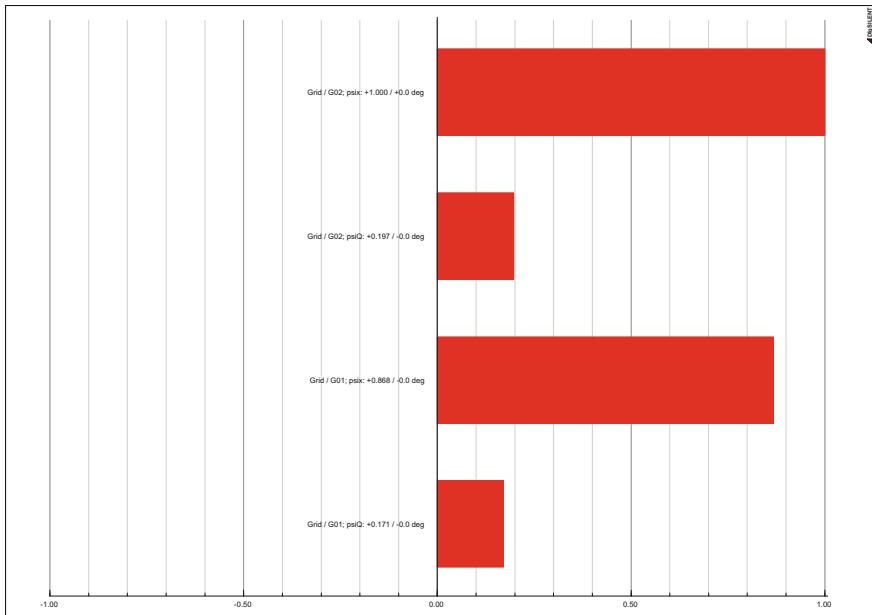


Fig. 9.33 Mode bar plots of the participation factors of the machines

Table 9.2 Designed local PSS parameters

T_w	T_1	T_2	T_3	T_4	k
10	0.05	0.015	0.08	0.01	10

Table 9.3 System eigenvalues; without PSS, with local PSS and WADC-PSS

Weakly damped eigenvalues	PSS type		
	No PSS	Local PSS only	WADC-PSS
$\lambda_{15}, \lambda_{16}$	$0.07 \pm j3.52$	$-0.33 \pm j3.14$	$-0.92 \pm j6.91$
$\lambda_{17}, \lambda_{18}$	$-0.64 \pm j6.82$	$-3.98 \pm j0.65$	$-4.54 \pm j0.35$
$\lambda_{24}, \lambda_{25}$	$-0.639 \pm j6.98$	$-2.09 \pm j5.98$	$-3.91 \pm j0.06$

Table 9.3 also includes the impact of the proposed local PSS on variation of the weakly damped modes. Although the local PSS units only stabilise the unstable modes and have a low impact on the weakly damped modes, the proposed WADC-PSS enhances the damping of all weakly damped modes, in addition to stabilising the unstable modes.

9.8 Conclusions

The use of wide-area measurement system (WAMS) has become an indispensable practical topic in monitoring and controlling the large-scale power system. WAMS architectures and mostly cited linear design techniques for proposing the wide-area damping controllers (WADCs) were introduced in this chapter. The role and the functions of PDC and PMU in WAMS were summarised. A WADC-PSS was designed based on modal analysis and using PSO method. WADC-PSS input signals are the non-local machine speed measurements from the WAMS.

Simulation of the proposed WADC-PSS and modal analysis of the system in DIgSILENT PowerFactory was discussed. Results confirm the improvement in damping of local and inter-area oscillatory modes through WADC-PSS.

References

1. H. Li, A. Bose, V.M. Venkatasubramanian, Wide-area voltage monitoring and optimization. *IEEE Trans. Smart Grid* **7**(2), 785–793 (2016)
2. S. Das, T.S. Sidhu, Application of compressive sampling in synchrophasor data communication in WAMS. *IEEE Trans. Industr. Inf.* **10**(1), 450–460 (2014)
3. J.F. Hauer, C.W. Taylor, Information, reliability, and control in the new power system, in *Proceedings of 1998 American Control Conference*, Philadelphia, PA, 24–26 June 1998
4. J.Y. Cai, Z. Huang, J. Hauer, K. Martin, Current status and experience of WAMS implementation in North America, in *Proceeding IEEE/Power Engineering Society Transmission and Distribution Conference Exhibition*, 2005, pp. 1–7
5. M. Shahraeini, M.H. Javidi, M.S. Ghazizadeh, Comparison between communication infrastructures of centralized and decentralized wide area measurement systems. *IEEE Trans. Smart Grid* **2**(1), 206–211 (2011)
6. A. Rendon Salgado, C.R. Fuerte Esquivel, J.G. Calderon Guizar, SCADA and PMU measurements for improving power system state estimation. *IEEE Lat. Am. Trans.* **13**(7), 2245–2251 (2015)
7. H.Y. Su, C.W. Liu, Estimating the voltage stability margin using PMU measurements. *IEEE Trans. Power Syst.* **31**(4), 3221–3229 (2016)
8. C. Queiroz, A. Mahmood, Z. Tari, SCADASim—a framework for building SCADA simulations. *IEEE Trans. Smart Grid* **2**(4), 589–597 (2011)
9. A.R. Messina, V. Vittal, D. Ruiz-Vega, G. Enriquez-Harper, Interpretation and visualization of wide-area PMU measurements using Hilbert analysis. *IEEE Trans. Power Syst.* **21**(4), 1763–1771 (2006)
10. A.G. Phadke, J.S. Thorp, *Synchronized Phasor Measurements and Their Applications* (Springer, New York, 2008)
11. J.D.L. Ree, V. Centeno, J.S. Thorp, A.G. Phadke, Synchronized phasor measurement applications in power systems. *IEEE Trans. Smart Grid* **1**(1), 20–27 (2010)
12. W. Jiang, V. Vittal, G.T. Heydt, A distributed state estimator utilizing synchronized phasor measurements. *IEEE Trans. Power Syst.* **22**(2), 563–571 (2007)
13. C. Lu, Y. Zhao, K. Men, L. Tu, Y. Han, Wide-area power system stabiliser based on model-free adaptive control. *IET Control Theory Appl.* **9**(13), 1996–2007 (2015)
14. D.J. Trudnowski, Estimating electromechanical mode shape from synchrophasor measurements. *IEEE Trans. Power Syst.* **23**(3), 1188–1195 (2008)

15. N.R. Chaudhuri, B. Chaudhuri, Damping and relative mode shape estimation in near real-time through phasor approach. *IEEE Trans. Power Syst.* **26**(1), 364–373 (2011)
16. A.D. Rajapakse, F. Gomez, K. Nanayakkara, P.A. Crossley, V.V. Terzija, Rotor angle instability prediction using post-disturbance voltage trajectories. *IEEE Trans. Power Syst.* **25**(2), 947–956 (2010)
17. Y. Li, D. Yang, F. Liu, Y. Cao, C. Rehtanz, *Interconnected Power Systems; Wide-Area Dynamic Monitoring and Control Applications* (Springer, Berlin, 2016)
18. X. Zhang, C. Lu, S. Liu, X. Wang, A review on wide-area damping control to restrain inter-area low frequency oscillation for large-scale power systems with increasing renewable generation. *Renew. Sustain. Energy Rev.* **57**, 45–58 (2016)
19. Y. Zhao, C. Lu, Y. Liu, Y. Han, Residue and identification based wide-area damping controller design in large-scale power system, in: *Proceeding of IEEE PES ISGT Conference*, 2012
20. J. Zhang, C.Y. Chung, Y. Han, A novel modal decomposition control and its application to PSS design for damping interarea oscillations in power systems. *IEEE Trans. Power Syst.* **27**(4), 2015–2025 (2012)
21. R. Vikhram Yohanandhan, L. Srinivasan, Decentralised wide-area fractional order damping controller for a large-scale power system. *IET Gener. Transm. Distrib.* **10**(5), 1164–1178 (2016)
22. X. Sui, Y. Tang, H. He, J. Wen, Energy-storage-based low-frequency oscillation damping control using particle swarm optimization and heuristic dynamic programming. *IEEE Trans. Power Syst.* **29**(5), 2539–2548 (2014)
23. G. Rogers, *Power System Oscillations* (Kluwer Academic Publishers, Dordrecht, 2000)
24. F.M. Gonzalez-Longatt, J.L. Rueda (eds.), *PowerFactory Applications for Power System Analysis* (Springer, Berlin, 2014)
25. DIgSILENT GmbH. *PowerFactory User's Manual. DIgSILENT PowerFactory Ver. 14.0* (DIgSILENT GmbH, Gomaringen, Germany, 2008)

Chapter 10

Optimal PMU Placement Framework Under Observability Redundancy and Contingency—An Evolutionary Algorithm Using DIgSILENT Programming Language Module



Mohsen Zare, Rasoul Azizipanah-Abarghooee, Mostafa Malekpour
and Vladimir Terzija

Abstract Over the last decade, there has been a considerable increase in deploying phasor measurement units (PMUs) in wide-area monitoring, protection, and control of power systems, as well as the development of smart transmission and distribution grid applications. This chapter is focused on the demonstration of capabilities of *DIgSILENT PowerFactory* software for solving the problem of optimal PMU placement in power networks. The optimal placement has been viewed from the perspective of satisfying the observability requirement of power system state estimator. Optimal placement of PMU is formulated as a practical design task,

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_10) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

M. Zare (✉)

Department of Electrical Engineering, Faculty of Engineering, Jahrom University,
Jahrom, Fars, Iran

e-mail: mzare@jahromu.ac.ir

R. Azizipanah-Abarghooee · V. Terzija

School of Electrical and Electronic Engineering, The University of Manchester,
Manchester M13 9PL, UK

e-mail: rasoul.azizipanah@manchester.ac.uk

V. Terzija

e-mail: vladimir.terzija@manchester.ac.uk

M. Malekpour

Department of Electrical Engineering, Faculty of Engineering, University of Isfahan,
Isfahan, Iran

e-mail: m.malekpour@eng.ui.ac.ir

considering some technical challenges like complete network observability, enough redundancy, and the concept of zero injection buses under PMU and tie-line critical contingencies. Furthermore, the meta-heuristic techniques on the basis of evolutionary computations are programmed as an optimisation toolbox in DIgSILENT Programming Language (DPL). A distinctive characteristic of the presented module is that the evolutionary algorithm is only coded in DPL without using the time-consuming process of interlinking *DIgSILENT PowerFactory* with another software package like MATLAB. In summary, the bus adjacency relationship matrix, zero injection bus (ZIB), observability in the presence of ZIB, and PMU/line contingencies are programmed in different DPLs and combined together with the DPL of an evolutionary algorithm to create the optimal PMU placement module. Also, the proposed toolbox is not case-dependent and can be run with the user-defined test systems, what is contributing to the proposed tool flexibility. Finally, the applicability and efficiency of the proposed optimal PMU placement module are investigated on the *DIgSILENT PowerFactory* version of IEEE 14- and 39-bus test systems.

Keywords DIgSILENT Programming Language · Evolutionary algorithm · Optimal PMU placement · Power system contingencies · Wide-area power system monitoring · Zero injection bus

10.1 Introduction

Over the past decade, smart grids have been designing and implementing wide-area monitoring system (WAMS) to provide the system operators more intelligent monitoring, control, and protection [1]. Hence, many researchers are motivated to technically deploy phasor measurement units (PMUs) to measure the essential data in few microseconds and transmit it to the long distances. As a result, the power utilities are switched to the new PMU-based wide-area measurement system from the conventional supervisory control and data acquisition (SCADA) technologies [2]. For instance, 1100 PMUs were installed across the US Eastern Interconnection area as per the year of 2014 which offers considerable transmission system coverage [3].

The process of locating PMUs in smart power systems is one of the big challenges imposing a high cost. Therefore, the utilities and researchers are interested to find a way to minimise this cost. Moreover, the PMU placement methodologies considering the dynamic criteria and phenomena are also provided. In general, there are two methodologies followed by power researchers for addressing the optimal PMU placement problem: (i) placement of PMUs to correctly represent critical dynamics and (ii) development of a list of places based on observability. The former method is used for the power system dynamics monitoring with a few PMUs [4, 5]. The dynamically developing coherent pictures of a power network are determined in this sense. This method is used to assess voltage security, analyse the voltage stability, and adapt the protection schemes [6, 7]. References [8–13] have deployed

the later strategy. It is to be noted that sitting the PMUs on each node of the grid results in a completely observed system [8]. Since a node is observable if the PMU is placed on it or installed on one or some of its neighbouring nodes, it is neither essential nor economical to implement such full installations. Consequently, the optimal PMU placement is raised as an economic optimisation problem. The optimal PMU placement problem was first introduced and formulated in [9]. The basic concept of PMU placement problem is formed according to the Kirchhoff current and voltage laws besides the concept of zero injection bus. However, some other issues like single contingencies of network lines and PMUs have not been evaluated in [9]. In general, two main methodologies comprise of mathematical-based and meta-heuristic-based algorithms that are developed for this problem. The mathematical ones are on the basis of preparing mathematical relationships and expressions [10]. Another method, i.e. meta-heuristic approaches, genetic algorithm [11], simulated annealing [12], chemical reaction [13] etc., are based on providing a creative process to find the global or near-global solution of the PMU placement problem. These methods can find a near optimal solution to the problem of PMUs placement so that the full network observability with the minimum number of PMUs is provided, which in turn mitigates the WAMS cost. However, in addition to the PMUs' cost, the number of unobservable buses in single PMU contingency, as well as the number of unobservable buses in single line contingency, is so important which are considered in this book chapter.

The power system test cases are usually modelled in *DIgSILENT PowerFactory* which is powerful software for power system simulation [14]. This well-known software implements a set of the script as command codes to facilitate the simulation procedure. These commands are coded in *DIgSILENT Programming Language* (DPL) part of PowerFactory, and they are applied to the system to append automation trend into the simulations. DPL scripts can arrange the parameters variation, running of diverse simulation, and commands. In this regard, the required data can be called from the database environment to the DPL script, and consequently, the analysis can be carried out to extract the simulation results. These superior capacities motivated us to utilise it in the practical PMU placement problem.

In this chapter, the optimal PMU placement problem based on the new *modified binary particle swarm optimisation* (MBPSO) algorithm is coded in a DPL script according to the following two main steps and some substeps. In the first step, the proposed optimisation technique is described and coded. The original particle swarm optimisation (PSO) algorithm has a simple but a speedy convergence ability which is applied extensively in power system optimisation problems and acquires approving results [15]. However, it may steadily stop generating the successful solutions through the population when it comes to highly complex nonlinear problems in electric power systems. Therefore, a powerful mutation strategy is added to the original structure to strongly improve the exploration and the exploitation processes of PSO. In the second stage, the objective functions and the constraints of the developed problem are formulated. It is to be noted that the network observability is assessed based on topological observability rules. To test and validate the implemented DPL-based PMU placement methodology, the IEEE

14- and 39-bus test systems are studied. DPL scripts and applications provided in this book chapter pave the way for the additional development of algorithms without using the time-consuming process of interlinking it with other software packages like MATLAB.

The outline of the proposed optimal PMU placement methodology and the configuration of calling each DPL scripts are shown in Figs. 10.1 and 10.2, respectively.

In Fig. 10.1, each section corresponds to a number. Following each number in ‘Description’ page of each DPL script can clarify how each step is implemented using DPL commands.

In Fig. 10.2, the blue box is considered as the master DPL script, and the red one is implemented to be as the slave sub-DPL. As can be seen, for example, ‘PMU_Calculation’ DPL script is called in ‘fitness’ DPL script while this calls two other DPL scripts during its implementation procedure.

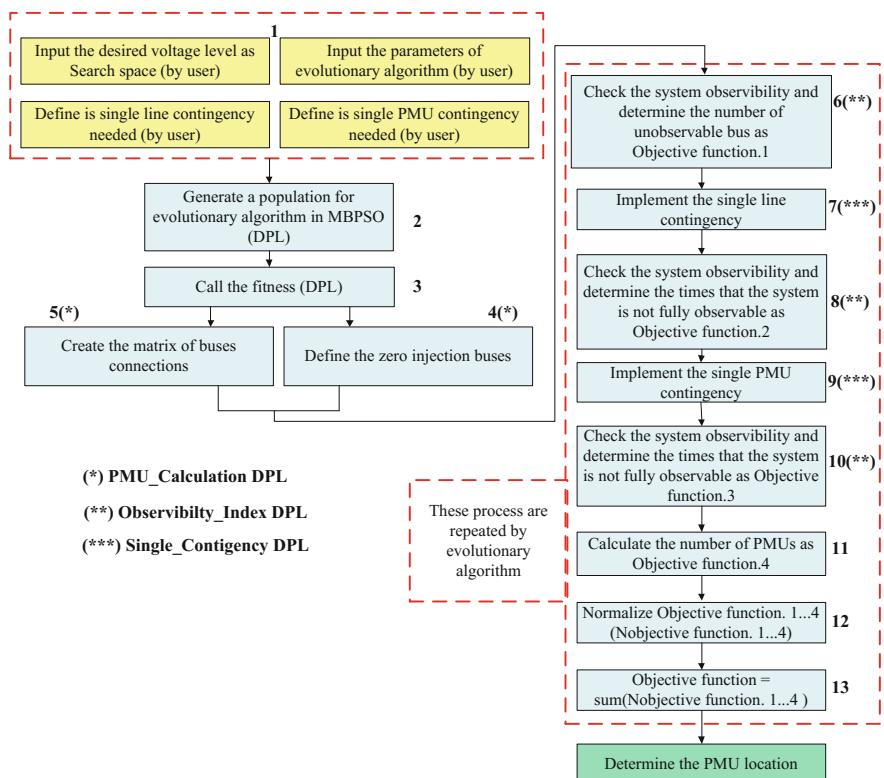


Fig. 10.1 The proposed optimal PMU placement methodology

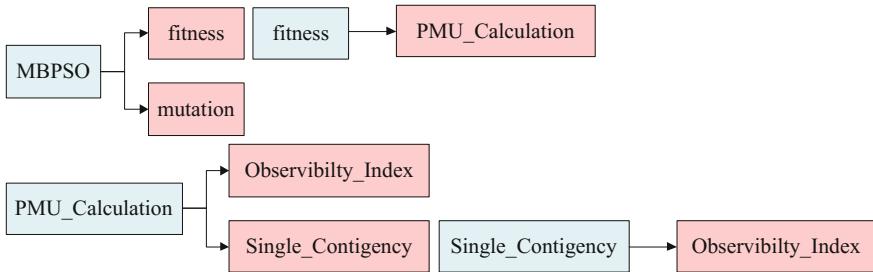


Fig. 10.2 Configuration of calling each DPL script

10.2 Basic Concept of PMU Placement Problem

The optimal PMU problem is defined as finding the location of these units to reach the full network observability with the minimum number of PMUs. Network observability has several rules which prevent the installation of PMUs overall network buses and help to minimise the cost of the optimisation problem. These rules are as follows [10–13, 16, 17]:

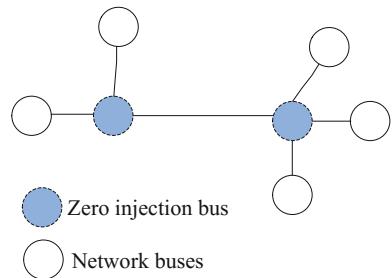
1. When a PMU is installed in a bus, the voltage phasor of target bus and the current phasor of all branches connected to this bus are observable. In this state, the bus is directly observable.
2. When the voltage phasor at one end of a branch and the current phasor of this branch are available, the voltage phasor at another end is also available.
3. The current phasor of the line can be calculated if the voltage phasor of both ends of this line is known.

It is to be noted that in order to implement the rule numbers 2 and 3, the target line impedance should be available.

4. If the current phasor of $n - 1$ lines connected to the zero injection bus (ZIB) is specified, the unknown current phasor can be calculated through the Kirchhoff's current law (KCL) equations. A bus is considered to be ZIB if it has no load or generator. Therefore, the sum of currents flowing to a ZIB is zero.
5. If the voltage phasor of all buses connected to the ZIB is known, node equation can be used to calculate the unknown voltage phasor of ZIB.
6. If a group of ZEBs is connected to Fig. 10.3, another rule can be extended for this condition. The node equation for these ZIBs is written as:

$$\sum_{j=1}^{N_{\text{br}}} Y_{ij}(V_i - V_j) = 0 \quad i = 1, 2, \dots, N_{\text{ZIB}} \quad (10.1)$$

Fig. 10.3 A set of connected zero injection buses



where N_{ZIB} is the number of ZIBs in a group, N_{br} is the number of branches in the group, V_i is the voltage phasor of bus i , and Y_{ij} is the line admittance between buses i and j . Here, N_{ZIB} complex equations are defined; if the voltages of all adjacent buses to the ZIBs are known, the voltages of ZIBs can also be calculated from these equations.

The rules mentioned above constitute the basic concept of finding the optimal location of PMUs in a power system.

In general, the developed optimal PMU placement problem is based on the following two main stages:

- (i) Optimisation technique (i.e. MBPSO),
- (ii) Calculating the objective function including the number of PMUs, number of unobservable buses in normal condition, number of unobservable buses in a single PMU contingency, and number of unobservable buses in a single line contingency.

10.3 DPL Script for Modified Binary Particle Swarm Optimisation

In order to handle the most power system optimisation problem, two types of approaches are usually used in the literature, i.e. Lagrange multiplier-based mathematical methods and population-based evolutionary algorithms (EAs). Although some of the algorithms of the first group are implemented by the system operator and planner because of their higher convergence speed and unique solution, they suffer from some shortcomings those are as follows:

- (i) Some of these algorithms usually implement the derivative operations during the optimisation process to give an optimal solution when the function is continuous, differentiable, and convex one. So the implementing of these methods is difficult.
- (ii) In some cases, linearisation of the objective functions and constraints may lead to the loss of accuracy.
- (iii) The dependency of these methods to initial conjecture and the weakness of some of them dealing with inequality constraints.

- (iv) These algorithms suffer from the handling of the complex optimisation problems such as NP-complete problem like optimal PMU placement problem.

The huge execution time and sensitivity to various parameters setting are the disadvantages of the EAs. However, this kind of methods is selected in this paper due to their simple implementation, their independence from the objective functions types, decision variable types, and initial solutions. Therefore, it is clear-cut that providing a robust and fast EA to reach the global or near-global optimal solution is an important task.

Based on this abstract, this section presents the DPL script of the proposed MBPSO method to solve the optimal PMU placement problem. The binary PSO (BPSO) is a well-known population-based optimisation algorithm which is proposed by Kennedy and Eberhart in [18]. As mentioned earlier, the original BPSO suffers from trapping in local optima when it is applied to combinatorial problems like the proposed PMU placement. In order to enhance the accuracy of the optimal solution and restrain the algorithm from a premature convergence, two new mutation strategies are appended into the original binary PSO. These mutation strategies have the capability to prevent agents from rapidly trapping in local optima, especially in the case of complex combinatorial problems. To implement the proposed strategies, in the first step, the following formulations are coded in DPL script as the formulation of original BPSO (population is similar in BPSO and MBPSO).

$$\mathbf{X}_j^{it} = [x_{1,j}^{it}, x_{2,j}^{it}, \dots, x_{i,j}^{it}, \dots, x_{N\text{var},j}^{it}]_{1 \times N\text{var}} \quad (10.2)$$

$$\text{MBPSO_pop}^{it} = [\mathbf{X}_1^{it}, \mathbf{X}_2^{it}, \dots, \mathbf{X}_j^{it}, \dots, \mathbf{X}_{N\text{pop}}^{it}]_{1 \times (N\text{var} \times N\text{pop})} \quad (10.3)$$

$$x_{p,i,j}^{it} = \mathbf{P}_\text{best}_{i,j}^{it} \oplus x_{i,j}^{it-1} \quad (10.4)$$

$$x_{g,i,j}^{it} = \mathbf{G}_\text{best}_i^{it} \oplus x_{i,j}^{it-1} \quad (10.5)$$

$$V_{i,j}^{it} = c_1 \otimes x_{p,i,j}^{it} + c_2 \otimes x_{g,i,j}^{it} \quad (10.6)$$

$$c_1, c_2 = \text{rand} \in \{0, 1\}$$

$$\sum_{i=1}^{N\text{var}} V_{i,j}^{it} \leq V_j^{\max} \quad (10.7)$$

$$x_{i,j}^{it} = V_{i,j}^{it} \oplus x_{i,j}^{it-1} \quad (10.8)$$

where \mathbf{X}_j^{it} is the vector of j th particle of the population at iteration it , $x_{i,j}^{it}$ is the i th variable of particle j at the iteration it , $\mathbf{P}_\text{best}_{i,j}^{it}$ is the i th variable of j th particle located in $\mathbf{P}_\text{best}^{it}$ matrix, $\mathbf{P}_\text{best}^{it}$ is the matrix contains the best position

encountered by each particle until iteration it , $G_{\text{best}}^{it}_i$ is the i th variable of the best position among all particles until iteration it (i.e. G_{best}^{it} matrix), $V_{i,j}^{it}$ is the i th variable of the velocity vector corresponds to the j th particle at iteration it , and V_j^{\max} is the maximum velocity vector corresponds to the j th particle. N_{var} and N_{pop} are the number of decision variables and initial population, respectively. Symbols \oplus , \otimes , and $+$ defines the *xor*, *and*, as well as *or* logics.

10.3.1 Initializing the MBPSO Algorithm in DPL Script

One of the main goals of DPL command is to provide an efficient environment for the users so that they can express their scripts and create some new programming codes as well as functions. The DPL command object which is called '*ComDpl*' is the fundamental part of each project to connect different input parameters, external objects, and variables to different functions or internal elements and to provide the output results or to change parameters. The DPL command objects provide users with an interfaced DPL script for the sake of configuration and preparation of useful codes for running the power system test case to reach the predefined objectives. Therefore, the DPL script runs a sequence of operation subroutines and accomplishes some functions' calculation. The definitions of different variables, parameters, sets and strings, their assignments, standard power system subroutines and functions, programming flow commands, and available instructions can cooperate through the DPL script of *DigSILENT PowerFactory* software [19]. A DPL command '*ComDpl*' can be produced deploying the '*New Object*' button in the data manager toolbar.

In order to code the MBPSO algorithm, a script using DPL is implemented. To this end, the number of decision variables should be firstly determined. Each bus with the nominal voltage equal or greater than the predefined threshold voltage is considered as the candidate bus (decision variable) for installing the PMU. The provided DPL script is portrayed in the below grey box. Several comments are appended to this code and other similar ones to explain their step-by-step procedure.

```

! ### MBPSO DPL script ###
double val0, val1, val2, val3, nvar, i, j,
    opt_fit, opt_loc, iter,best_fit, best_loc, NPMU,
    U, NBUS, cnt_V, Vmax, c1, c2, best_fit_tmp, Initializing, Ts, Tf;
set Term;
object O;
string nme;
Ts = GetTime(4);
LDF.Execute();                                !Load flow calculation is executed
ClearOutput();
NOIE.Init(1,1);                               !NOIE:matrix that contains the number of
                                              !evaluation the 'Observability_Index' DPL
script
! ### Define number of network buses (nvar) ###
Term=AllRelevant('*.*ElmTerm');
O=Term.First();
NBUS=0;      !NBUS: Number of network buses in desired voltage range (decision variable)
while(O){ s
    U=O::uknom;
    if(U>Un.and.O:iUsage==0.and.O:outserv==0){ !Un=threshold voltage level
        NBUS+=1;
    }
O=Term.Next();}
nvar=NBUS;                                     !nvar:number of decision variables
BEST.Init(1,nvar+1);                           !BEST:best result matrix
BEST_plot.Init(iter_max,2);                     !BEST_plot:best result in each iteration

```

In order to generate the initial population of the MBPSO algorithm namely MBPSO_pop¹, it and P_best¹ matrices are initialized with the same size. The last column in these two matrices is devoted to the objective function value. In addition, the initial velocity of each particle is set to zero. Then, a random number is generated corresponding to each decision variable. If this number is greater than 0.8, the target decision variable will be rounded to 1. Otherwise it would be 0. It means that the probability of PMU installation in a bus is considered to be less than 20%. However, it depends on to the system operator preference and can be changed accordingly. In the following, the initialization process is coded in the DPL script.

```

! ###Generating the initial population ###
pop.Init(n_pop,nvar+1);                      !pop (i.e. MBPSO_pop):population matrix of
                                              !evolutionary Particles in MBPSO algorithm algorithm
P_best.Init(n_pop,nvar+1);                    !P_best:matrix of best encountered position by
                                              !each particle
V.Init(n_pop,nvar);                          !V:matrix of particles velocity
Initializing=0;                             !Initializing:see section (1.5)
for(i=1;i<=n_pop;i+=1){
    for(j=1;j<=nvar;j+=1){
        val0=fRand(0);
        if(val0>0.8){
            pop.Set(i,j,1);
            P_best.Set(i,j,1);
        }
    }
}

```

After determining all variables, the ‘fitness’ DPL script will be called according to the explanation provided in Sect. 10.4 and a specific DPL script. At first, the location of the best particle (i.e. ‘best_loc’) which its fitness function (‘best_fit’) is

the best among all population is set to one. Once each member of the population has been generated and its fitness function has been computed, the values of ‘*best_loc*’ and ‘*best_fit*’ must be updated in consequence. It is to be noted that in the process of population initialization, the matrix of ‘*pop*’ and the best encountered position by each particle, ‘*P_best*’, are similar.

In order to release the constraints of full observability under normal and contingency conditions, the penalty factor will be added to the fitness function of each particle that is not satisfied. On the other hand, if the particle is initialized in such a way that the PMUs are located on all the network buses, the full observability in normal and contingency conditions will be achieved. Thus, the objective function with the bigger value than the number of network buses defines the unacceptable particle, and the target particle is renewed. This procedure can be implemented besides the selection of low probability of installing the PMU (less than 20%) in order to obtain the full observability conditions with the fewer number of PMUs in initializing the process. A short script is shown below in which it is possible to see the construction of the above-mentioned process.

```

fitness.Execute(i,nvar,Un,co1,co2,co3,co4,Initializing,ln_cntgy,PMU_cntgy);
Initializing=1;
  if(i==1){
    best_loc=1;                                !best_loc:location of best particle
    best_fit=pop.Get(i,nvar+1);                  !best_fit:best value for objective function
  }
best_fit_tmp=pop.Get(i,nvar+1);          !best_fit_tmp: an ancillary variable
  if(i>1.and.best_fit_tmp<best_fit){
    best_loc=i;
    best_fit=best_fit_tmp;
  }
val0=pop.Get(i,nvar+1);                  !Get the value of objective function of
                                         !particle "i"
P_best.Set(i,nvar+1,val0);
  if(val0>NBUS){i=1;}
}

```

Finally, the particle with the best fitness function is saved in ‘*BEST*’ matrix as the best solution until now and its fitness function value is stored in ‘*BEST_plot*’ matrix. This matrix is considered for plotting the convergence characteristic of the suggested algorithm. Also, each column of ‘*BEST*’ matrix is labelled by bus names which are read from ‘*A_mat*’ matrix (see Sect. 10.5). In the following, the developed DPL script is provided.

```

NPMU=0;
for(j=1;j<=nvar+1;j+=1){           !Set the best particle to BEST matrix
    val0=pop.Get(best_loc,j);
    if(j<=nvar){
        nme=A_mat.RowLbl(j);
        BEST.Collbl(nme, j);
        NPMU+=val0;
    }
    BEST.Set(1,j,val0);
}
BEST_plot.Set(1,1,1);BEST_plot.Set(1,2,best_fit);

```

10.3.2 DPL Script for Evolutionary Process

The MBPSO algorithm deploys an iterative evolutionary process to update the particles using the following DPL script.

```

for(iter=1;iter<=iter_max;iter=iter+1){ !iter_max: maximum iteration in evolutionary
                                         !process

```

In order to update the position of each particle and calculate the corresponding fitness function, the equations set 10.4–10.8 can be followed based on the below explanations.

1. The velocity of each variable is calculated as follows: The j th variable of the i th particle from the ‘MBPSO_pop ^{it} ’, ‘P_{best} ^{it} ’, and ‘G_{best} ^{it} ’ will be called. Afterwards, the Eqs. (10.4) and (10.5) are employed to calculate the values of $x_{p,i,j}^{it}$ and $x_{g,i,j}^{it}$, respectively.
2. Then, the MBPSO coefficients of c_1 and c_2 are generated randomly. These values are deployed in Eq. (10.6) to calculate the updated value of velocity for each variable (i.e. $V_{i,j}^{it}$).
3. In the implemented DPL script which is illustrated below, the value of the V_j^{\max} is considered to be equal to the number of assigned PMUs to each particle (the summation of all variables in each particle). In the proposed approach, if the corresponding velocity of each particle is greater than V_j^{\max} (i.e. ‘cnt_V > Vmax’ in the DPL script), various elements of the velocity vector which is set to one must be selected randomly and substituted with zero.

The following DPL script provides the aforementioned three steps.

```

! ###Calculating the velocity vector and updating the position of each particle ###
for(i=1;i<=n_pop;i+=1){
  Vmax=0;                                !Vmax:maximum allowable velocity for each particle
  cnt_V=0;                                !cnt_V=counter for velocity of each particle
  for(j=1;j<=nvar;j+=1){
    val0=pop.Get(i,j);                    !Updating the velocity of each particle
    Vmax+=val0;
    val1=P_best.Get(i,j);
    val2=BEST.Get(1,j);
    val1=val0.eor.val1;
    val2=val0.eor.val2;
    c1=fRand(0);
    c1=round(c1);
    c2=fRand(0);
    c2=round(c2);
    val1=c1.and.val1;
    val2=c2.and.val2;
    val0=val1.or.val2;
    V.Set(i,j,val0);
    cnt_V+=val0;
  }
  while(cnt_V>Vmax){
    val0=fRand(0);
    val0=val0*(nvar-1)+1;
    val0=round(val0);
    val0=v.Get(i,val0);
    if(val0==1){
      v.Set(i,val0,0);
      cnt_V-=1;
    }
  }
}

```

4. After that, the position of each particle can be updated using Eq. (10.8). Finally, the values of objective functions can be updated based on the instructions presented in Sect. 10.4. The suggested DPL script is presented hereinafter.

```

for(j=1;j<=nvar;j+=1){
  val0=pop.Get(i,j);
  val1=v.Get(i,j);
  val0=val0.eor.val1;          !Updating the position of each variable in each
                               !particle
  pop.Set(i,j,val0);
}
fitness.Execute(i,nvar,Un,co1,co2,co3,co4,Initializing,ln_cntgy,PMU_cntgy);
  !ln(PMU)_cntgy 1:single line(PMU) contingency is considered (0: do not consider)
}

```

10.3.3 Implementation of the First and Second Mutation Strategies

The meta-heuristic methods such as the original BPSO suffering from the premature convergence and the entrapping in local optima. To cope with this drawback, the

proposed MBPSO algorithm puts the mutation strategy together with the BPSO to escape from local optima, and eventually halt to an acceptable solution.

In the developed algorithm, two efficient strategies are suggested. The first mutation strategy is shown in (10.9) which is a local search technique around the best-obtained solution while (10.10) defines a local search around each particle.

(A) *First mutation operator*

$$x_{i,r}^{it} = \begin{cases} 1 & \text{if } k_i = 1 \text{ and } \text{rand}_i() \geq 0.2 \\ 0 & \text{if } k_i = 1 \text{ and } \text{rand}_i() < 0.2 \\ G_best_i^{it} & \text{if } k_i = 0 \end{cases} \quad i = 1, \dots, Nvar; \quad (10.9)$$

$$\sum_{i=1}^{Nvar} k_i = \text{mut_gen}$$

with k_i and $\text{rand}_i()$ as the rounded and original random number in the range of $[0, 1]$ which are generated for the i th variable of the r th particle, respectively. This particle is randomly selected from the existing population. Also, there is one more limitation that forces the summation of the k_i , $i = 1, \dots, Nvar$ variables to be equal to ‘*mut_gen*’. This parameter is set to five in the proposed mutation strategy. It means that according to (10.9), the value of one and zero will be substituted with the i th variable of solution r with $k_i = 1$, if the random number $\text{rand}_i()$ is more or equal to 0.2 and less than 0.2, respectively. Otherwise, $x_{i,r}^{it}$ will be replaced with the i th element of the best solution. It means that this mutation method tries to install the PMUs in different network buses. This process is called the local search around the ‘*BEST*’ solution. As can be seen, if ‘*CI*’ equals to zero, this mutation strategy is neglected. In the following, the DPL script of this mutation strategy is provided.

```
! ###Implementing the first proposed mutation method ###
    !Add the best solution to population randomly
val0=fRand(0);
val0=val0*(n_pop-1)+1;
val0=round(val0);
for(i=1;i<=nvar*C1+1*C1;i+=1){
    val1=BEST.Get(1,i);
    pop.Set(val0,i,val1);
}
    !Local search around the best population
for(i=1;i<=mut_gen*C1;i+=1){!mut_gen= A predefined variable in range [1,nvar]
val1=fRand(0);
val1=val1*(nvar-1)+1;
val1=round(val1);
val2=fRand(0);
if(val2>0.2){
    pop.Set(val0,val1,1);
}
else{pop.Set(val0,val1,0);}
}
```

$$\begin{aligned}
 x_{i,j}^{mut,it} = & \begin{cases} x_{i,j}^{it} & \text{if } x_{i,j}^{it} = 1 \text{ and } \text{rand}_{i,j}() > mut_val \\ 0 & \text{if } x_{i,j}^{it} = 1 \text{ and } \text{rand}_{i,j}() \leq mut_val \\ 0 & \text{if } x_{i,j}^{it} = 0 \end{cases} \\
 & \text{if } f(\mathbf{X}^{mut,it}(x_{i,j}^{mut,it})) < f(\mathbf{X}^{it}(x_{i,j}^{it})) : \mathbf{X}^{it}(x_{i,j}^{it}) \leftarrow \mathbf{X}^{mut,it}(x_{i,j}^{mut,it}) \\
 & j = 1, 2, \dots, N\text{pop}; \quad i \in \{1, 2, \dots, N\text{var}\}
 \end{aligned} \tag{10.10}$$

In this local search mutation method, an $N\text{var}$ -by-2 matrix namely ‘rnd_mat’ is constituted in such a way that the number of all network buses and random numbers in the range of [0, 1] are located in the first and second columns, respectively. Then, this matrix is sorted in descending order based on the second column. Figure 10.4 shows how the network buses are sorted randomly. Here, according to the first two lines of (10.10), the i th variable of solution j with $x_{i,j}^{it} = 1$ will be substituted with zero or maintained unchanged, if the random number $\text{rand}_{i,j}()$ is less or equal to ‘ mut_val ’ and more than it, respectively. It should be noted that the index i is selected based on the second column of ‘rnd_mat’ matrix. Additionally, there is one more restriction on the first line of (10.10) that the target particle shall not penalise in the process of calculating the objective function. It is to be noted that the parameter ‘ mut_val ’ is set to 0.8, so that each variable ‘1’ will be replaced by ‘0’ with the probability of 0.8. This procedure is done if ‘ $C2$ ’ equals to 1.

The DPL script of the above-mentioned local search mutation approach is provided below.

Fig. 10.4 Sorting the network buses in descending order

Random Number	Bus Number	Random Number	Bus Number
0.33	1	0.024	2
0.024	2	0.18	3
0.18	3	0.33	1
0.42	4	0.42	4

Before sorting After sorting

```

rnd_mat.Init(nvar,2);
for(i=1;i<=n_pop;i+=1){
    for(j=1;j<=nvar;j+=1){
        rnd_mat.Set(j,1,j);
        val0=fRand(0);
        rnd_mat.Set(j,2,val0);
    }
}
rnd_mat.SortToColumn(2);
val0=pop.Get(i,nvar+1);
if(val0<=NBUS*C2){
    for(j=1;j<=nvar;j+=1){
        val1=rnd_mat.Get(j,1);
        val2=pop.Get(i,val1);
        if(val2==0) continue;
        val2=fRand(0);
        if(val2>mut_val) continue;
        pop.Set(i,val1,0);
        fitness.Execute(i,nvar,un,co1,co2,co3,co4,Initializing,ln_cntgy,PMU_cntgy);
}
}

```

For the replacement operation, the fitness function of the new solution $\mathbf{X}^{mut,it}\left(x_{i,j}^{mut,it}\right)$ should be compared with the fitness function of the existing vector $\mathbf{X}^{it}\left(x_{i,j}^{it}\right)$ according to the last line of (10.10). If the new particle provides the better fitness function, i.e. $f\left(\mathbf{X}^{mut,it}\left(x_{i,j}^{it}\right)\right) < f\left(\mathbf{X}^{it}\left(x_{i,j}^{it}\right)\right)$, the new one is kept and the process for other variables will be done, else, the target variable is changed to ‘1’ and the process will be continued. The process proposed in (10.9) and (10.10) is called the first mutation strategy.

In the following, the DPL script of this process is shown.

```

val2=pop.Get(i,nvar+1);
if(val0<val2){      !val0(previous present) value of objective function
    pop.Set(i,val1,1);
    pop.Set(i,nvar+1,val0);
}
else{val0=pop.Get(i,nvar+1);}
}

```

– Updating of ‘P_best’ matrix

The process of updating the best encountered solution by each particle (i.e. ‘P_best’) is accomplished at the end of each loop on the basis of the following DPL script.

```

for(i=1;i<=n_pop;i+=1){
    val0=pop.Get(i,nvar+1);
    val1=P_best.Get(i,nvar+1);
    if(val0<val1){
        for(j=1;j<=nvar+1;j+=1){
            val0=pop.Get(i,j);
            P_best.Set(i,j,val0);
        }
    }
}

```

– *Updating of ‘BEST’ and ‘BEST_plot’ matrices*

For the replacement operation, the fitness function of the new solution generated in the current iteration *it* should be compared with the fitness function of the best solution obtained in previous iteration *it – 1*. If the fitness function is better than that of the best solution, the global best particle will be replaced with this solution in ‘BEST’ matrix. Moreover, the best objective function in each iterate of the algorithm is also saved in ‘BEST_plot’ for convergence plot purpose. These procedures are implemented in the below DPL script.

```

opt_fit=BEST.Get(1,nvar+1);
if(best_fit<opt_fit){           !NPMU=an ancillary variable to count the number of PMUs
    NPMU=0;
    for(j=1;j<=nvar+1;j+=1){
        val0=pop.Get(best_loc,j);
        NPMU+=val0;
        BEST.Set(1,j,val0);
    }
    NPMU-=val0;
}
opt_fit=BEST.Get(1,nvar+1);
BEST_plot.Set(iter+1,1,iter+1);BEST_plot.Set(iter+1,2,opt_fit);

```

(B) *Second mutation operator*

It is to be noted that due to the complexity of the optimal PMU placement problem and presence of the local optima around the global ones, premature convergence may be occurred. To cope with this drawback, another powerful mutation strategy is implemented which not only keeps the population diversity but also adapted to the manner of first mutation strategy.

For this mutation technique, the below ‘Mutation’ DPL script is executed.

```
mutation.Execute(nvar,n_pop,mut_rate);
```

The detail of this ‘*Mutation*’ DPL script is provided as follows:

```
! ###Mutation DPL script###
double val0, val1, val2, i;
val0=nvar*n_pop;
val0=val0*mut_rate;           !mut_rate:A predefined variable in range [0,1]
for(i=1;i<=val0;i+=1){
    val1=fRand(0);
    val1=val1*n_pop;
    val1=round(val1);
    val1=max(val1,1);
    val2=fRand(0);
    val2=val2*nvar;
    val2=round(val2);
    val2=max(val2,1);
    pop.Set(val1,val2,1);
}
```

In second mutation strategy, some variables of particles are set to ‘1’. The number of variables which will be changed is controlled by ‘*mut_rate*’ parameter which is set to 0.35. As can be seen, The number of candidate PMUs is increased in this mutation strategy while the unnecessary ones are removed in Eq. (10.10).

The process of implementing the proposed MBPSO algorithm is shown in Fig. 10.5 step-by-step.

In Fig. 10.5 each section is corresponded to a number. Following each number in ‘*Description*’ page of each DPL scripts can clarify how each step is implemented using DPL commands.

10.3.4 DPL Script for Calculating the Objective Function

At first, the population is imported from the ‘MBPSO’ DPL script and then another DPL namely ‘*fitness*’ DPL script is called to compute the below formulated objective functions (10.11)–(10.14). Afterwards, the aggregated objective function is added to the end column of population matrix that is ‘*pop*’ as presented in below DPL script.

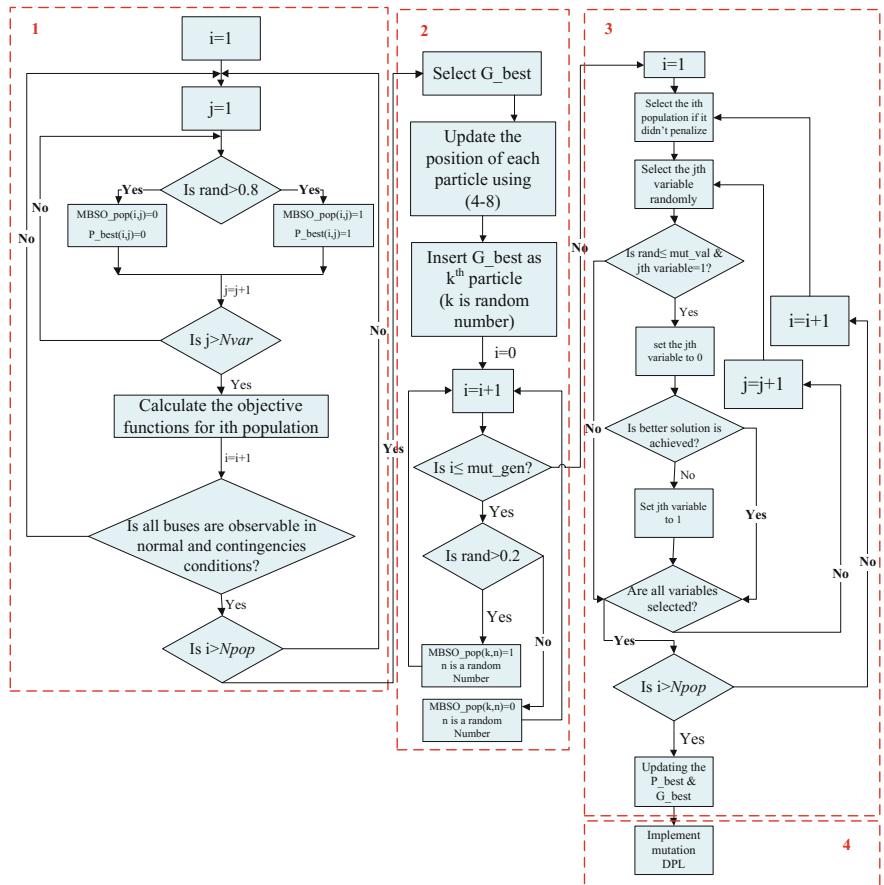


Fig. 10.5 Flow chart of implementing the proposed MBPSO

```

! #####fitness DPL #####
double col, val0, obj1, obj2, obj3, obj4, T_obs_net_LNE,
       T_obs_net_PMU, T_OBS, objective_function;
pop_cost.Init(1,nvar);
for(col=1;col<=nvar;col+=1){
val0=pop.Get(i,col);
pop_cost.Set(1,col,val0);   !pop_cost:pop for calculating the objective functions
}
PMU_Calculation.Execute(Un,Initializing,ln_cntgy,PMU_cntgy,T_obs_net_LNE,
T_obs_net_PMU,T_OBS);
!Un:Thersholt voltage range   !T_obs_net_LNE(PMU):No. of un-observable bus in single
                               !line (PMU) contingency condition
                               !T_OBS: No. of un-observable bus in normal condition
                               !Calculating the objective functions
obj1=0;                      !1st objective function = Number of PMU
obj2=0;                      !2nd objective function
obj3=0;                      !3rd objective function
obj4=0;                      !4th objective function
for(col=1;col<=nvar;col+=1){
val0=pop_cost.Get(1,col);
obj1+=val0;
}
obj2=T_obs_net_LNE;
obj3=T_obs_net_PMU;
obj4=T_OBS;
objective_function=obj1*co1+obj2*co2+obj3*co3+obj4*co4;
pop.Set(i,nvar+1,objective_function);
                           !devoting the objective function value to particle "i"

```

More precise investigation of DPL script of ‘MBPSO’ shows that any change in the population is accomplished in a loop with the index of ‘ i ’. This index is also imported to the ‘fitness’ DPL script. Here, the i th row of the population matrix is saved in ‘ pop_cost ’. This matrix is imported to other DPL scripts which will be explained hereinafter. Then, the ‘*PMU_calculation*’ DPL script will be executed. The output variables of this DPL script can be used to calculate the aggregated objective function. The summation of all variables in each particle which is located in ‘ pop_cost ’ matrix is the first objective while the number of unobservable buses in the single line contingency, single PMU contingency, and normal conditions are the second, third, and fourth objective functions, respectively. The expressions (10.11)–(10.14) illustrate different parts of the aggregated objective function. These values are multiplied by their weighting factors, and consequently, their summation is equal to the aggregated objective function. The coefficients can be considered as the penalty factor. These penalty factors can convert the constrained objective function to an unconstraint augmented one according to the following equations:

$$\text{obj}_1 = \sum_{i=1}^{N\text{var}} x_{i,j} \quad (10.11)$$

$$\begin{aligned} \text{obj}_2 &= \sum_{k=1}^{N\text{line}} \sum_{n=1}^{N\text{bus}} \text{OBS_LNE}_{n,k} \\ \text{OBS_LNE}_{n,k} &= \begin{cases} 1 & \text{if bus } n \text{ is unobservable when line } k \text{ is out of service} \\ 0 & \text{if bus } n \text{ is observable when line } k \text{ is out of service} \end{cases} \end{aligned} \quad (10.12)$$

$$\begin{aligned} \text{obj}_3 &= \sum_{k=1}^{\text{obj}_1} \sum_{n=1}^{N\text{bus}} \text{OBS_PMU}_{n,k} \\ \text{OBS_PMU}_{n,k} &= \begin{cases} 1 & \text{if bus } n \text{ is unobservable when PMU } k \text{ is out of service} \\ 0 & \text{if bus } n \text{ is observable when PMU } k \text{ is out of service} \end{cases} \end{aligned} \quad (10.13)$$

$$\begin{aligned} \text{obj}_4 &= \sum_{n=1}^{N\text{bus}} \text{OBS}_n \\ \text{OBS}_n &= \begin{cases} 1 & \text{if bus } n \text{ is unobservable in normal condition} \\ 0 & \text{if bus } n \text{ is observable in normal condition} \end{cases} \end{aligned} \quad (10.14)$$

10.4 PMU_Calculation DPL Script

The ‘PMU_calculation’ DPL script is provided to define the ZIB and connectivity network matrix. The zero injection and nonzero injection buses of the network can be defined based on the following explanation.

As shown in this DPL script, the number of buses which their voltages are in the desired range is counted as explained before. The process of this DPL script can be continued if the value of parameter ‘*Initializing*’ is zero. It is to be pointed out that once the ‘*fitness*’ DPL script is called for the first time, this parameter is zero (see Sect. 10.3.1) and then it is set to one. The reason is that all matrices calculated in ‘PMU_calculation’ DPL script are constant in the evolutionary process, thoroughly. To determine the nonzero injection bus, the generated power, the load power (i.e. general load or motor load), the compensated power, and the external network power of each bus can be checked; if one of these values is greater than zero, this bus is added to the set of nonzero injection buses. In the following DPL script, this procedure is completely explained.

```

! ###Defining the zero and non-zero injection bus ###
set Term, S, BUS, Line, NZRObus, ZRObus, Tr2, Tr3, Swth;
object O, O1, bus1, bus2, bus3;
double NUM, U, i, index, val0, val1, val2, r, c, k, nrow, cunt, j, obs_net;
string nme, nme1, nme2, nme3;
Term=AllRelevant('*.*ElmTerm');
Term.SortToName(0);
O=Term.First();
NUM=0;
while(O){
  U=O:e:uknom;
  if(U>=Un.and.O:iUsage==0.and.O:outserv==0){      !Selecting bus bur with nominal
                                                       !voltage greater
                                                       !than desired voltage
    NUM+=1;
  }
O=Term.Next();
}
if(Initializing==0){
A_mat.Init(NUM,NUM);           !A_mat:network connectivity matrix
ZERO_LOC_BUS.Init(NUM,1);       !ZERO_LOC_BUS:location of zero injection bus
O=Term.First();
while(O){
  U=O:e:uknom;
  if(U>=Un.and.O:iUsage==0.and.O:outserv==0){
    BUS.Add(O);                  !Add the bus bur with nominal voltage greater than
                                   !desired voltage to 'BUS' set
    S=O.GetConnectedElms();
    val0=O:m:Pgen;
    val1=O:m:Qgen;              !NZRObus:set of non-zero injection bus
      if(val0>0.or.val1>0){NZRObus.Add(O);}
    val0=O:m:Pmot;
    val1=O:m:Qmot;
      if(val0>0.or.val1>0){NZRObus.Add(O);}
    val0=O:m:Pload;
    val1=O:m:Qload;
      if(val0>0.or.val1>0){NZRObus.Add(O);}
    val0=O:m:Pcomp;
    val1=O:m:Qcomp;
      if(val0>0.or.val1>0){NZRObus.Add(O);}
    val0=O:m:Pnet;
    val1=O:m:Qnet;
      if(val0>0.or.val1>0){NZRObus.Add(O);}
}
}

```

Additionally, the bus which is connected to another node through two- or three-winding transformers and at the same time, the voltage magnitude of the secondary or thirdly bus is under the desired voltage, can be appended into the set of nonzero injection buses. After this process, the buses which are not in the set of nonzero injection bus are added to the ‘ZERO_LOC_BUS’ matrix as the ZIBs. The procedure of completing the set of nonzero injection buses besides adding the component to ‘ZERO_LOC_BUS’ matrix is presented in the following DPL scripts.

```

01=S.First();
  while(01){
    val0=01.IsClass('ElmTr2');
    if(val0){
      if(01:t:utrn_1<Un){
        NZRObus.Add(0);
        continue;
      }
    }
    val0=01.IsClass('ElmTr3');
    if(val0){
      if({01:t:utrn3_m<Un.or.01:t:utrn3_l<Un}.and.01:t:utrn3_l>0){
        NZRObus.Add(0);
        continue;
      }
    }
    01=S.Next();
  }
}
0=Term.Next();
}
BUS.SortToName(0);
0=BUS.First();
NUM=1;
while(0){
  val0=0;
  val1=NZRObus.IsIn(0);
  if(val1){
    ZRObus.Add(0);
    val0=1;
  }
  ZERO_LOC_BUS.RowLbl(0:e:loc_name, NUM);
  ZERO_LOC_BUS.Set(NUM,1,val0);
! #####Creating the network connectivity matrix #####
  A_mat.RowLbl(0:e:loc_name, NUM);
  A_mat.Collbl(0:e:loc_name, NUM);
  0=BUS.Next();
  NUM+=1;
}

```

The second section of the ‘PMU_calculation’ DPL script is devoted to the procedure of creating the matrix ‘A_mat’ as the network connectivity matrix as indicated in the following DPL script. To construct the network connectivity matrix, each series element in the *DIGSILENT PowerFactory* should be checked to find the connected terminals. In this regard, the rows and the columns of ‘A_mat’ matrix are labelled with the name of buses based on the above DPL script.

Afterwards, all the network lines are chosen and the connectivity of each element to the target line is checked. Indeed, the connected buses to the target line are saved as follows:

```

!Buses connected to lines
Line=AllRelevant('*.ElmLne');
O=Line.First();
while(O) {
  S=O.GetConnectedElms();
  val0=1;
  val1=1;
  O1=S.First();
  while(O1){
    val2=O1.IsClass('*.ElmTerm');
    if(val2=1.and.val0=1){bus1=O1;val0=0;val2=0;}
    if(val2=1.and.val1=1){bus2=O1;val1=0;val2=0;}
    if(val0=0.and.val1=0){break;}
    O1=S.Next();
  }
  nme1=bus1:e:loc_name;
  nme2=bus2:e:loc_name;
}

```

Once the buses of both sides of the line are specified, the names of these buses can be compared to the row (column) label of matrix ‘A_mat’. When the locations of the target buses are realised in the ‘A_mat’ matrix, the corresponding arrays of this matrix must be set to one.

- If the selected line is located between buses ‘i’ and ‘j’, the array in row ‘i’ and column ‘j’ and the array in row ‘j’ and column ‘i’ need to be set to one. If there are other connections between these two buses, the corresponding arrays of these buses should be consequently changed in ‘A_mat’ matrix. It should be pointed out that if the voltage magnitude of this selected line is out of the desired voltage range, the name of this line cannot be found in the row (or column) label of ‘A_mat’ matrix. Thus, this line cannot be appended into the mentioned matrix. This process is clearly explained in the following DPL script.

```

nrow=A_mat.NRow();
r=0;
c=0;
for(i=1;i<=nrow;i+=1){
  nme=A_mat.Rowlbl(i);
  val0=strcmp(nme,nme1);
  val1=strcmp(nme,nme2);
  if(val0=0){r=i;}
  if(val1=0){c=i;}
  if(r>>0.and.c>>0){
    cunt=A_mat.Get(r,c);
    A_mat.Set(r,c,1+cunt);
    A_mat.Set(c,r,1+cunt);
    break;
  }
  o=Line.Next();
}

```

- Two-winding transformers are other elements those can link two buses together. In this condition, if the voltage amplitude of the low-voltage side is greater than the desired voltage level, the above-mentioned procedure can be repeated. Otherwise, the low-voltage side of the transformer is considered to be a load.
- Three-winding transformers are other elements which can connect the network buses to each other. Since these transformers have three voltage levels. Thus, three discrepant scenarios must be examined. If the voltage amount of the low-voltage side of these transformers is greater than the desired voltage level, three voltage levels are appended into the matrix of ‘A_mat’. Also, if the desired voltage level is located between the voltage magnitude of the low- and medium-voltage side, the high- and the medium-voltage sides are appended into the ‘A_mat’.
- Switches are other devices that have this capability to connect two network buses whenever they are close. Therefore, the closed switches are also added to the ‘A_mat’ matrix.

Once the ‘A_mat’ matrix is completed for all the above-mentioned elements, the arrays of this matrix are copied to the matrix of ‘A_mat_aux’ and then it is edited based on the provided DPL script. In this regard, if the array of ‘A_mat’ matrix be equal or greater than ‘1’, the corresponding array of ‘A_mat_aux’ matrix set to ‘1’.

Finally, two other DPL scripts are executed (as below grey box) and explained from now on.

```
Observability_Index.Execute(NUM,obs_net,T_OBS);
Single_Contingency.Execute(NUM, ln_cntgy,PMU_cntgy,T_obs_net_LNE,T_obs_net_PMU);
```

10.5 DPL Script for Observability_Index

One of the main challenges of the optimal PMU placement problem is the investigation of full network observability under the proposed condition. The ‘*Observability_Index*’ DPL script calculates the network observability, and it is explained in the following three subsections.

10.5.1 First Section: Network Observability Based on Rules 1–5

To investigate the network observability index, some steps should be taken. First of all, all the system buses which they or one of their incident buses is equipped with PMU are observable. To check this circumstance, the ‘pop’ as well as the ‘A_mat’ matrices can be deployed. In this regard, if the i th column of the ‘pop’ matrix is equal to ‘1’ (i.e. one PMU is located at bus i), all connected buses to this bus are

considered as observable bus and the array of the i th row of ‘ OBS ’ matrix must be set to one. The DPL script of this process is provided as follows:

```
! ###Investigating the network observability ###
double val0, val1, obs_index, NZCB, col_i,
       cunt1, cunt2, cunt3, stp_crtra;
int row, col;
cunt1=NOIE.Get(1,1);cunt1+=1;NOIE.Set(1,1,cunt1); !Times that this DPL script is
                                                       !executed
                           !If a PMU be connected to bus i
                           !all connected bus to bus i is observable
OBS.Init(NBUS,1); C_OBS.Init(NBUS,1); !OBS: Network observability matrix
for(row=1;row<=NBUS;row+=1){
  val0=pop.Get(1,row);
  if(val0){
    for(col=1;col<=NBUS;col+=1){
      val0=A_mat.Get(row,col);
      if(val0) {OBS.Set(col,1,1);C_OBS.Set(col,1,1);}
    }
  }
}
```

In the first step, if the summation of all arrays of the ‘ OBS ’ matrix is equal to the number of network buses (which have the desired voltage range), the full network observability is achieved; else the second step should be implemented.

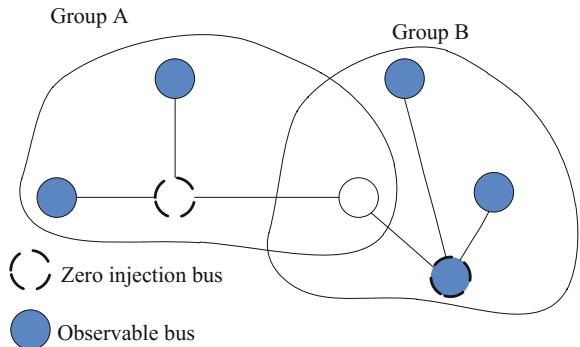
In the second step, all ZIBs are checked and if all buses connected to the considered ZIB are observable, this bus is also observable. In this regard, each ZIB will be selected and the number of buses connected to this bus will be counted using the following DPL script.

```
! ###Implementation of rules 4 & 5###
else{
  stp_crtra=1;
  while(stp_crtra<>0){
    stp_crtra=0;
    for(row=1;row<=NBUS;row+=1){
      val0=ZERO_LOC_BUS.Get(row,1);
      if(val0==0) continue;
      cunt1=0;
      for(col=1;col<=NBUS;col+=1){
        cunt2=A_mat.Get(row,col);
        cunt1+=cunt2;           !number of bus connected to zero injection bus
      }
    }
  }
}
```

After that, the number of observable buses connected to this bus can be counted as follows:

```
cunt3=0;
for(col=1;col<=NBUS;col+=1){
  val0=A_mat.Get(row,col);
  if(val0){
    cunt2=OBS.Get(col,1);
    cunt3+=cunt2;           !number of observable bus connected to zero
                           !injection bus
  }
}
```

Fig. 10.6 Two groups of buses with zero injection bus



If all buses consist of ZIB and its connected buses are observable except one of them, the ‘*OBS*’ array corresponded to this bus is found and set to one.

The process of implementation of the rules 4 and 5 is repeated until one array of ‘*OBS*’ matrix changes. The reason for this fact is shown in Fig. 10.6.

As can be seen in Fig. 10.6, there are two ways to start the observability checking process. If the buses of group ‘A’ are firstly investigated, the observability of all buses cannot be confirmed; however, if the buses of group ‘B’ are firstly checked and continued with those of group ‘A’, the observability of all buses can be justified. Thus, the observability process is repeated until each array of ‘*OBS*’ matrix is changed. This process is illustrated in below DPL script in which the value of ‘*stp_crtra*’ controls it. If any array of ‘*OBS*’ matrix is changed, the value of ‘*stp_crtra*’ is set to 1 and the above process will be repeated.

```

if(cunt3=cunt1-1){ !all buses connected to zero injection bus is
!observable except one of them
val0=OBS.Get(row,1);
if(val0){
  for(col=1;col<=NBUS;col+=1){
    val0=A_mat.Get(row,col);
    if(val0=0) continue;
    val0=OBS.Get(col,1);
    if(val0=0){OBS.Set(col,1,1);
    stp_crtra+=1;
    }
  }
  else{
    OBS.Set(row,1,1);
    stp_crtra+=1;
  }
}
if(cunt3=cunt1){C_OBS.Set(row,1,0);}
}
}
obs_index=0;
for(row=1;row<=NBUS;row+=1){ !Check the observability of all buses
val0=OBS.Get(row,1);
obs_index+=val0;
}
if(obs_index==NBUS){obs_net=1;}!All network buses are observable

```

The second section of the ‘*Observability_Index*’ DPL is explained in the next subsection.

10.5.2 Second and Third Sections: Network Observability Based on Rule 6

The DPL script of rule 6 is shown below. It is to be noted that the ‘*else*’ at the beginning of this code is pertinent to the command of ‘*if*’ at the last line of previous DPL script. Indeed, once the full network observability cannot be achieved by implementing of rules 4 and 5 of Sect. 10.2, the provided rule 6 should be investigated. To do this, ZIBs are selected from the ‘*ZERO_LOC_BUS*’ matrix. While each ZIB is found, ‘*ZRO_Con_Bus*’ matrix will be initialized and the buses connect to this ZIB are checked. In this step, ‘*ZRO_Con_Bus*’ matrix will be contained of the target ZIB and all buses connected to this node. The process of defining the all buses connected to a target zero injection bus is based on the below DPL script.

```
! #####Second section of Observability_Index DPL #####
else{
    stp_crtra=1;
    while(stp_crtra<>0){
        stp_crtra=0;
        for(row=1;row<=NBUS;row+=1){
            val0=ZERO_LOC_BUS.Get(row,1); !ZERO_LOC_BUS:Location of zero injection bus
            if(val0>0) continue;
            ZRO_Con_Bus.Init(NBUS);
            for(col=1;col<=NBUS;col+=1){
                val0=A_mat.Get(row,col);
                ZRO_Con_Bus.Set(col,1,val0); !ZRO_Con_Bus:matrix contains all buses connected
                                            !to zero injection buses
            }
        }
    }
}
```

The ZIBs connected to the target bus and the number of its ZIBs should also be determined. The matrix ‘*NZCB_mat*’ is assigned to this matter and is initialized to save the number of ZIBs which are connected together. Each array of ‘*ZRO_Con_Bus*’ matrix will be checked, if the value of the selected array is one and corresponded to the ZIB from the ‘*ZERO_LOC_BUS*’ matrix, all connected bus to the new ZIB are added to the ‘*ZRO_Con_Bus*’ matrix. Here, it is supposed that the loop index is 6 and the third array of ‘*ZRO_Con_Bus*’ is set to one. Since the index loop is greater than 3, so this new bus cannot be investigated, while the bus 3 may be a ZIB. Due to this reason, if the array of ‘*ZRO_Con_Bus*’ matrix corresponded to a ZIB changes from zero to one while the loop index is greater than the index of this target array, the loop index should be reset to one. The variable ‘*cunt2*’ is an ancillary variable which helps to control this process, and it is changed when the criteria mentioned above are satisfied. This procedure is explained in the below DPL script.

```

NZCB_mat.Init(NBUS,1);           !NZCB_mat:matrix contains zero injection Bus
                                !connected to each other
    for(col=1;col<=NBUS;col+=1){
        val0=ZRO_Con_Bus.Get(col,1);
        if(val0=0) continue;
        val0=ZERO_LOC_BUS.Get(col,1);
        if(val0=0) continue;
        NZCB_mat.Set(col,1,1);
        cunt2=0;
        for(col_i=1;col_i<=NBUS;col_i+=1){
            val0=A_mat.Get(col,col_i);
            if(val0){
                cunt1=ZRO_Con_Bus.Get(col_i,1);
                val0=ZERO_LOC_BUS.Get(col_i,1);
                if(cunt1=0.and.col_i<=col.and.val0=1){cunt2=-1;}
                ZRO_Con_Bus.Set(col_i,1,val0);
            }
        }
        if(cunt2=-1){col=1;}
    }
}

```

Once the ‘*ZRO_Con_Bus*’ matrix has been completed, the sum of arrays of this matrix will define the number of buses in a group of ZIBs which are connected together (Fig. 10.6) and the summation of ‘*NZCB_mat*’ matrix’s arrays shows the number of ZIB in the group. Calculating the number of zero injection bus and the total number of buses in a group of connected zero injection buses is done based on the following DPL script.

```

NZCB=0;
    for(col_i=1;col_i<=NBUS;col_i+=1){
        val0=NZCB_mat.Get(col_i,1);
        NZCB+=val0;           !NZCB: number of ZIBs connected to each other
    }
    cunt1=0;
    cunt2=0;
    for(col=1;col<=NBUS;col+=1){
        val0=ZRO_Con_Bus.Get(col,1);
        if(val0){
            cunt1+=1;
            val0=C_OBS.Get(col,1);
            val1=ZERO_LOC_BUS.Get(col,1);
            cunt2+=val0;
        }
    }
}

```

Since each ZIB can construct a node Eq. (10.1), so that if all group’s buses are observable except one or more ZIB, these buses should be also considered as the observable buses. The criterion of ‘*cunt1-NZCB<=cunt2*’ shows this situation in the below provided DPL script. For example, if a group of ZIBs has eight buses and only three of them are ZIB, at least, five buses should be observable to complete the observability of this group. It is notable that the observability of all buses of this group will be checked on the basis of ‘*C_OBS*’ matrix (i.e. based on rules 1–3 provided in Sect. 10.2) to construct enough node Eq. (10.1) to achieve the phasors of unobservable buses. It should be mentioned that if all connected buses to a ZIB are observable, this bus cannot construct a useful node Eq. (10.1). When all buses

connected to a ZIB are observable, its corresponding array in ‘*C_OBS*’ matrix is set to zero (i.e. ‘*if(cunt3=cuntI){C_OBS.Set(row,I,0);}*’). The implementation of this concept can be found in the following DPL script.

```

if(cunt1-NZCB<=cunt2){
    for(col=1;col<=NBUS;col+=1){
        val0=ZRO_Con_Bus.Get(col,1);
        if(val0){
            val1=OBS.Get(col,1);
            if(val1=0){stp_crtra+=1;}
            OBS.Set(col,1,1);
        }
    }
}

```

The final substep of second stage is to check the network observability by counting the number of observable buses using the following DPL script.

```

obs_index=0;
for(row=1;row<=NBUS;row+=1){
    val0=OBS.Get(row,1);
    obs_index+=val0;
}
if(obs_index==NBUS) {obs_net=1;}           !All network buses are observable
}

```

The third stage of ‘*Observability_Index*’ DPL script is provided. What should be underscored is that the observability of a bus cannot be calculated if the target bus is isolated from the network; thus, the array relating to this bus should be set to one in ‘*OBS*’ matrix as shown in below DPL script.

```

! ###Third section of Observability_Index DPL ###
if(obs_net=0){
    for(row=1;row<=NBUS;row+=1){
        val0=OBS.Get(row,1);
        cunt1=0;
        if(val0=0){
            for(col=1;col<=NBUS;col+=1){
                val0=A_mat.Get(row,col);
                cunt1+=val0;
            }
            if(cunt1=1){OBS.Set(row,1,1);} !Isolated bus from the network
        }
    }
}

```

After which, the network observability can be checked and the number of unobservable buses will be calculated according to the following DPL script.

```

obs_index=0;
for(row=1;row<=NBUS;row+=1){
    val0=OBS.Get(row,1);
    obs_index+=val0;
}
if(obs_index==NBUS){obs_net=1;} !All network buses are observable
}
T_OBS=0;                                !T_OBS= Number of un-observable bus
for(row=1;row<=NBUS;row+=1){
    val0=OBS.Get(row,1);
    T_OBS+=val0;
}
T_OBS=NBUS-T_OBS;

```

10.6 DPL Script for Single Contingency Conditions (Single_Contingency DPL)

The $n-1$ contingency is usually sufficient to avert losing the network observability, since a PMU or a single line outage may lead to observability losing of a few number of buses. Also, in some scenarios which the radial buses are only connected to another network bus, obtaining the network observability using more than two PMUs (i.e. one of them is installed in the radial node and the second one is installed in its adjacent node) is not possible. Furthermore, the investment restriction is another challenge which avoids the PMU installation on the power system network beyond the actual requirement. It is to be noted that the proposed approach is generic enough to be applied on other higher-order contingencies.

As the network observability should be checked in a single PMU or line contingency, a ‘*Single_Contingency*’ DPL script is developed to check the contingency condition. As mentioned earlier, each array of ‘*A_mat*’ matrix (i.e. $A_mat(i, j) \in \{0,1\}$) defines a connection between bus i and j . Additionally, the array (i, j) is equal to the array (j, i) . Thus, the single line contingency is implemented by setting the arrays (i, j) and (j, i) to zero if they have the value of one. Then, the ‘*Observability_Index*’ DPL script is called to investigate the network observability in this condition. Meanwhile, if the full network observability cannot be achieved, a penalty factor (equal to the number of network buses) is considered as the number of unobservable buses in the single line contingency objective (i.e. ‘*T_obs_net_LNE*’) and then the process is ended. Note that the ‘*A_mat*’ matrix returns to its initial. It should be noted that the users can calculate the number of unobservable buses without the penalty value by commenting the section of ‘*if (T_OBS>0){}*’.

The DPL script for implementing the single line contingency concept is as follows:

```

! ###Single_Contingency DPL ###
!Single line contingency
!T_obs_net_LNE=Number of un-observable bus in single
!line contingency
if(ln_cntgy==1){
    for(i=1;i<=NUM;i+=1){           !NUM=NBUS
        for(j=i+1;j<=NUM;j+=1){
            val0=A.Get(i,j);
            if(val0==0.or.val0>=2) continue;
            A_mat.Set(i,j,0);
            A_mat.Set(j,i,0);
            Observability_Index.Execute(NUM,obs_net,T_OBS);
            T_obs_net_LNE=T_obs_net_LNE+T_OBS;
            A_mat.Set(i,j,1);
            A_mat.Set(j,i,1);
            if(T_OBS>0){
                i=NUM;
                j=NUM;
                T_obs_net_LNE=NUM;
            }
        }
    }
}

```

The process of single PMU contingency is started if the value of '*T_obs_net_LNE*' is equal to zero, otherwise, the value of '*T_obs_net_PMU*' will be set to a penalty value as the number of unobservable buses in single PMU contingency objective. To simulate the single PMU contingency condition, each variable of the MBPSO population which has the value of one will be set to zero. After that, the '*Observability_Index*' DPL script is called and the network observability in this condition is checked. A penalty value is considered as the '*T_obs_net_PMU*' variable, if full network observability is not achieved. While this process is similar to the single line contingency, a simple change is needed to calculate the number of unobservable buses in the single PMU contingency condition. The DPL script is provided as follows:

```

!Single PMU contingency
!T_obs_net_PMU=Number of un-observable bus in single
!PMU contingency
if(PMU_cntgy==1{
    if(T_obs_net_LNE==0){
        for(i=1;i<=NUM;i+=1){
            val0=pop.Get(1,i);
            if(val0==0) continue;
            pop.Set(1,i,0);
            Observability_Index.Execute(NUM,obs_net,T_OBS);
            T_obs_net_PMU=T_obs_net_PMU+T_OBS;
            pop.Set(1,i,1);
            if(T_OBS>0){
                i=NUM;
                j=NUM;
                T_obs_net_PMU=NUM;
            }
        }
    }
    else {T_obs_net_PMU=30;}
}

```

10.7 Simulation Results

This section presents the results of implementing the created DPL scripts on IEEE 14- and 39-bus test systems in order to evaluate the accuracy of the suggested method. Both the IEEE 14-bus and 39-bus test systems are simulated in the *DIGSILENT PowerFactory* software, and their data are taken from [20]. Some input parameters can be changed by users based on his/her preferences. Parameters of MBPSO algorithm are ‘*n_pop*’, ‘*iter_max*’, ‘*iter_max*’, ‘*mut_gen*’, ‘*mut_val*’, and ‘*mut_rate*’ which are explained in previous sections.

In the first step to validate the effectiveness of the proposed modification over the original BPSO algorithm, the obtained results of MBPSO algorithm by solving the optimal PMU placement problem those are the minimum number of PMUs which can observe the system and placement results are listed in Table 10.1. To reach the optimal number of PMUs, it is required to deploy the maximum potential of the ZIBs. In fact, an extension of the PMU placement rules is done to achieve the network full observability using the fewer number of PMUs. In this regard, if the initial population of evolutionary algorithm keeps their diversity during the evolutionary process, a different combination of PMUs location can be tested, and the effectiveness of these rules is highlighted, while the premature convergence decreases the effectiveness of PMU placement rules by decreasing the number of feasible solutions.

It should be noted that the results are achieved considering the single contingency of PMUs or network lines. The results of proposed MBPSO are less than the original BPSO and the recently developed approach in [21] which confirm the effectiveness of proposed modifications.

Additional information about two test systems including the number of zero injection bus and their locations is also shown in Table 10.1.

In order to clarify the effectiveness of each proposed modifications, the problem is solved in two different cases while one modification is considered in each case and other is neglected. Table 10.2 shows the results of this comparison in different 20 independent trial runs. As can be seen from this table, mutation strategy related to (10.10) has a great effect on achieved results. Furthermore, Eq. (10.9) besides the second mutation strategy can enhance the ability and robustness of algorithm compared to original one. Investigating the initial population after the evolutionary process confirms population diversity while the population of BPSO algorithm deals with trapping in local optima and premature convergence. All in all, it should be regarded that however the proposed mutation strategies increase the CPU time, but using these strategies the number of initial population can be decreased, and the better solutions are achieved in a fewer iterations of the evolutionary process.

Moreover, the number of installed PMUs for single PMU contingency, single line contingency, and without contingency for IEEE 14-bus and 39-bus systems are (7, 7, 3) and (17, 12, 8), respectively. It is clear-cut that considering contingencies leads to a growing of the number of installed PMUs compared to the base case circumstance. It is obvious that the number of PMUs in the case of single line

Table 10.1 Best results for two test systems

Solution methodologies	Test system	Location of ZIB	Number of installed PMU	PMU location
Original BPSO	NE 39-bus	1, 2, 5, 6, 9, 10, 11, 13, 14, 17, 19, 22	25	3, 4, 6, 7, 9, 13, 14, 15, 16, 18, 20, 21, 23, 25, 26, 27, 29, 32, 33, 34, 35, 36, 37, 38, 39
	IEEE 14-bus	7	7	2, 4, 5, 6, 9, 11, 13
MBPSO	NE 39-bus	1, 2, 5, 6, 9, 10, 11, 13, 14, 17, 19, 22	17	3, 7, 8, 12, 13, 16, 20, 21, 23, 25, 26, 29, 30, 34, 36, 37, 38
	IEEE 14-bus	7	7	1, 2, 4, 6, 9, 10, 13
Reference [21]	NE 39-bus	1,2,5,6,9,10,11, 13,14,17,19,22	18	3, 4, 8, 16, 20, 23, 25, 26, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38
	IEEE 14-bus	7	8	1, 2, 4, 6, 8, 9, 11, 13

Table 10.2 Comparison of different proposed mutation strategies

Test system	Scenario	Best	Worst	STD
NE 39-bus	Without Eq. (10.9)	17	18	0.3
	Without Eq. (10.10)	24	30	1.76
IEEE 14-bus	Without Eq. (10.9)	7	7	0
	Without Eq. (10.10)	7	10	0.73

contingency has less value with respect to other contingency types. It is to be noted that a PMU needs to be considered in each radial node in the case of single line contingency since this bus will be islanded after the outage of the line connected to it so that the installation of PMU is completely essential.

In the other hand, the PMU contingency is equivalent to lack of an observable source, while the line outage means the loss of an observable path. Therefore, the PMUs which are installed in the single line outage are less than those of the single PMU loss. In Table 10.3, the optimal number and allocated PMUs for both power networks are tabulated under two types of contingencies. The results of the proposed method are also compared with Ref. [21] which is shown that the number of PMUs obtained by the proposed MBPSO method in both cases is less than them. The suggested mutation strategy can provide a balance between the diversity and convergence rate or diversification and intensification capacities of the MBPSO which is an apparently critical requirement in providing a good quality solution.

It is to be noted that the achieved locations for PMUs are not unique, so other different results with the same number of PMUs and disparate locations can be

Table 10.3 Number and location of PMUs achieved by suggested technique in different contingencies and without it

		Single PMU contingency		Single line contingency		Without contingency	
		Number of PMUs	Location of PMUs	Number of PMUs	Location of PMUs	Number of PMUs	Location of PMUs
Test system	Solution methodologies						
IEEE 14-bus	Proposed MBPSO	7	1, 2, 4, 6, 9, 11, 13	7	1, 2, 4, 6, 9, 10, 13	3	2, 6, 9
	Reference [21]	7	1, 2, 4, 6, 9, 11, 13	7	1, 3, 6, 8, 10, 12, 14	3	2, 6, 9
NE 39-bus	Proposed MBPSO	17	3, 6, 8, 13, 16, 20, 21, 23, 25, 26, 29, 30, 32, 34, 36, 37, 38	12	3, 6, 8, 13, 16, 20, 22, 23, 25, 26, 29, 30	8	3, 8, 13, 16, 20, 23, 25, 29
	Reference [21]	17	2, 3, 6, 8, 10, 12, 16, 20, 21, 23, 25, 26, 29, 34, 36, 37, 38	16	8, 10, 16, 18, 24, 26, 28, 30, 31, 32, 33, 34, 35, 36, 37, 38	8	8, 11, 16, 18, 20, 23, 25, 29

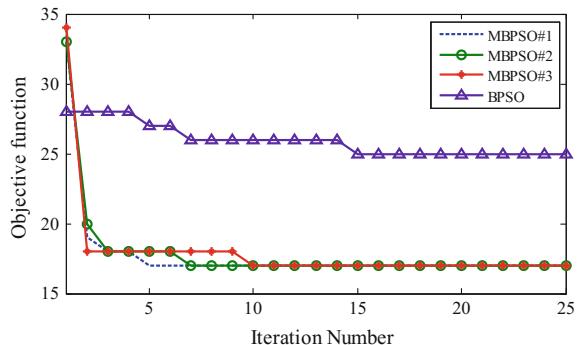
extracted from the proposed optimisation method. Different results can be considered as different scenarios, and the system planner can select the solution which is more suitable from the point of view of topology implementation, etc.

Furthermore, the proposed MBPSO and PSO are applied to the optimal PMU placement problem in different 20 independent trial runs, and the statistical data including the best, the worst, and the standard deviation of the achieved results are presented in Table 10.4.

Table 10.4 Statistical analysis for best results

Solution methodologies	Test system	Best	Worst	STD
Original BPSO	NE 39-bus	25	34	2.67
	IEEE 14-bus	7	10	1.01
MBPSO	NE 39-bus	17	18	0.22
	IEEE 14-bus	7	7	0

Fig. 10.7 Convergence characteristics for 39-bus test system



The small values of the standard deviations corresponding to two cases confirm the robustness of the proposed algorithm.

It is noteworthy that apart from being robust and fast, the suggested algorithm can converge to the near-global optimum within 5, 7, and 10 iterations for three different runs, as portrayed in Fig. 10.7. Moreover, the convergence characteristic of original BPSO is also illustrated in this figure. One can observe that the MBPSO can quickly converge to an optimal and robust solution. It should be noted that the best solution of IEEE 14-bus system is usually achieved in initial iterations.

10.8 Conclusion

A new module is developed for the application of evolutionally algorithms e.g. MBPSO on power system optimisation problems like optimal PMU placement using DPL command. The DPL scripts are designed for coding the proposed MBPSO algorithm as well as objective function. Due to the complexity of the problem, two new and effective modification processes have been added to the proposed algorithm to enhance its exploitation and exploration, simultaneously. The proposed PMU placement formulation includes four terms in objective function those are: (1) number of PMUs, (2) number of unobservable buses in the normal conditions, (3) number of unobservable buses in single line contingency, and (4) number of unobservable buses in single PMU contingency. Each part is completely introduced by its corresponding DPL scripts. The effectiveness, flexibility, and scalability of the suggested algorithm are verified by the statistical and convergence analysis as well as experimental studies. The major findings are as follows:

- (1) Six efficient rules are introduced to check the network observability and to prevent the installation of PMUs across the whole network buses and help to minimise the cost of the optimisation problem.

- (2) A new and robust evolutionary algorithm using DPL module is designed for *DIGSILENT PowerFactory*.
- (3) A flexible toolbox without using the time-consuming process of interlinking *DIGSILENT PowerFactory* with another software package like MATLAB is presented.
- (4) The designed module is flexible and generic enough so that all DPL scripts' setting can be changed by users for different scenarios, test systems, and goals. It means that it is not case-dependent and can be run with the user-defined test systems.

References

1. M. Qiu, W. Gao, M. Chen, J.W. Niu, L. Zhang, Energy efficient security algorithm for power grid wide area monitoring system. *IEEE Trans. Smart Grid*. **2**(4), 715–723 (2011)
2. R. Sodhi, M.I. Sharieff, Phasor measurement unit placement framework for enhanced wide-area situational awareness. *IET Gener. Transm. Distrib.* **9**(2), 172–182 (2015)
3. "North American synchrophasor initiative," U.S. Department. Energy, Washington, DC, USA, Tech. Rep., 2014. [Online]. Available: <https://www.naspi.org>
4. I. Kamwa, A.K. Pradhan, G. Joos, S.R. Samantaray, Fuzzy partitioning of a real power system for dynamic vulnerability assessment. *IEEE Trans. Power Syst.* **24**(3), 1356–1365 (2009)
5. M. Dehghani, B. Shayanfar, A.R. Khayatian, PMU ranking based on singular value decomposition of dynamic stability matrix. *IEEE Trans. Power Syst.* **28**(3), 2263–2270 (2013)
6. A. Pal, G.A. Sanchez-Ayala, V.A. Centeno, J.S. Thorp, A PMU placement scheme ensuring real-time monitoring of critical buses of the network. *IEEE Trans. Power Del.* **29**(2), 510–517 (2014)
7. E.E. Bernabeu, J.S. Thorp, V.A. Centeno, Methodology for a security/dependability adaptive protection scheme based on data mining. *IEEE Trans. Power Del.* **27**(1), 104–111 (2012)
8. I. Kamwa, M. Leclerc, D. McNabb, Performance of demodulation-based frequency measurement algorithms used in typical PMUs. *IEEE Trans. Power Del.* **19**(2), 505–514 (2004)
9. T.L. Baldwin, L. Mili, M.B. Boisen, R. Adapa, Power system observability with minimal phasor measurement placement. *IEEE Trans. Power Syst.* **8**(2), 707–15 (1993)
10. M. Esmaili, K. Gharani, H.A. Shayanfar, Redundant observability PMU placement in the presence of flow measurements considering contingencies. *IEEE Trans. Power Syst.* **28**(4), 3765–3773 (2013)
11. F. Aminifar, C. Lucas, A. Khodaei, M. Fotuhi-Firuzabad, Optimal placement of phasor measurement units using immunity genetic algorithm. *IEEE Trans. Power Del.* **24**(3), 1014–1020 (2009)
12. R.F. Nuqui, A.G. Phadke, Phasor measurement unit placement techniques for complete and incomplete observability. *IEEE Trans. Power Del.* **20**(4), 2381–2388 (2005)
13. M.H. Wen, J. Xu, V.O. Li, Optimal multistage PMU placement for wide-area monitoring. *IEEE Trans. Power Syst.* **28**(4), 4134–4143 (2013)
14. F. Gonzalez-Longatt, and J. L. Rueda, "PowerFactory applications for power system analysis," Springer, Dec. 2014
15. Y. Del Valle, G.K. Venayagamoorthy, S. Mohagheghi, J.C. Hernandez, R.G. Harley, Particle swarm optimization: basic concepts, variants and applications in power systems. *IEEE Trans. Evol. Comput.* **12**(2), 171–195 (2008)

16. M. Hajian, A.M. Ranjbar, T. Amraee, B. Mozafari, Optimal placement of PMUs to maintain network observability using a modified BPSO algorithm. *Int. J. Electr. Power Energy Syst.* **33** (1), 28–34 (2011)
17. V. Terzija, G. Valverde, D. Cai, P. Regulski, V. Madani, J. Fitch, S. Skok, M.M. Begovic, A. Phadke, Wide-area monitoring, protection, and control of future electric power networks. *Proc. IEEE*. **99**(1), 80–93 (2011)
18. J. Kennedy, and R.C. Eberhart, A discrete binary version of the particle swarm algorithm, in *In Systems, Man, and Cybernetics 1997, Computational Cybernetics and Simulation, 1997 IEEE International Conference*, vol. 5, (1997), pp. 4104–4108
19. Factory DP, in *PowerFactory User's Manual*. DIgSILENT GmbH Version 15.0 (2015)
20. MatPower Software Package. [Online]. Available: <http://www.pserc.cornell.edu/matpower>
21. K.G. Khajeh, E. Bashar, A.M. Rad, G.B. Gharehpetian, Integrated model considering effects of zero injection buses and conventional measurements on optimal PMU placement. *IEEE Trans. Smart Grid*. (2016)

Chapter 11

Implementation of Slow Coherency Based Controlled Islanding Using DIgSILENT PowerFactory and MATLAB



I. Tyuryukanov, M. Naglič, M. Popov
and M. A. M. M. van der Meijden

Abstract Intentional controlled islanding is a novel emergency control technique to mitigate wide-area instabilities by intelligently separating the power network into a set of self-sustainable islands. During the last decades, it has gained an increased attention due to the recent severe blackouts all over the world. Moreover, the increasing uncertainties in power system operation and planning put more requirements on the performance of the emergency control and stimulate the development of advanced System Integrity Protection Schemes (SIPS). As compared to the traditional SIPS, such as out-of-step protection, ICI is an adaptive online emergency control algorithm that aims to consider multiple objectives when separating the network. This chapter illustrates a basic ICI algorithm implemented in PowerFactory. It utilises the slow coherency theory and constrained graph partitioning in order to promote transient stability and create islands with a reasonable power balance. The algorithm is also capable to exclude specified network branches from the search space. The implementation is based on the coupling of Python and MATLAB program codes. It relies on the PowerFactory support of the Python scripting language (introduced in version 15.1) and the MATLAB Engine for Python (introduced in release 8.4). The chapter also provides a

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_11) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

I. Tyuryukanov (✉) · M. Naglič · M. Popov · M. A. M. M. van der Meijden
Intelligent Electrical Power Grids, Department of Electrical Sustainable Energy,
Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: i.tyuryukanov@tudelft.nl

M. A. M. M. van der Meijden
TenneT TSO B.V., Utrechtseweg 310, 6812 AR Arnhem, The Netherlands

case study to illustrate the application of the presented ICI algorithm for wide-area instability mitigation in the PST 16 benchmark system.

Keywords Intentional controlled islanding · Controlled network separation · Generator coherency · Graph partitioning · Python scripting · DIgSILENT PowerFactory to MATLAB interface via Python

11.1 Introduction

Due to the electrical industry deregulation and massive grid integration of renewable energy sources, electric power systems (EPSs) are expected to operate close to their stability limits. Several large-scale blackouts during the recent decades [1, 2] have demonstrated the increased vulnerability of existing electric power grids. Due to these reasons, blackout prevention has become a topic of great importance for the operation of future EPS.

Intentional controlled islanding (ICI) is an adaptive wide-area protection algorithm belonging to the class of System Integrity Protection Schemes (SIPS) [3]. The basic idea is to define in an existing network a set of islands so that the initial disturbance, which could lead to a system collapse, remains confined within one of the islands. Studies of historical blackouts (e.g. [4]) show that a proper system islanding combined with load shedding and generator dropping has the potential to prevent wide-area blackouts. Compared to traditional SIPS, such as out-of-step protection, ICI is a real-time control technique which in general aims to consider multiple objectives, e.g. load-generation balance, generator coherency, transmission line availability, thermal limits, voltage stability and transient stability [5, 6]. Due to this highly adaptive and sophisticated nature, the design of ICI algorithms is currently an active research area. To enable a near real-time situational awareness, an ICI algorithm requires access to Wide-Area Measurement Systems (WAMSs) data.

An ICI method should be used as the last measure to rescue the power grid from a dangerous instability. Consequently, the overall ICI problem is commonly subdivided into two stages: *when to island* and *how to island* [7]. The first stage aims to promptly determine the “point of no return” after which only ICI can save the grid. The second stage aims to split the network in a way that results in a stable islanded operation with all restoration constraints satisfied. In this chapter, some important considerations regarding the second global challenge are presented, together with the implementation of a simple ICI algorithm using PowerFactory and MATLAB.

Among the multiple ICI objectives listed above, only the following ones will be considered in the simplified algorithm presented in this chapter, namely: generator coherency, transmission line availability, load-generation balance (in an indirect way) and transient stability (in an indirect way). The resulting solution promotes transient stability of islands by putting only coherent generators into each island and by reducing the changes in generators’ electric power through cutting transmission lines with a small active power flow. At the same time, the opening of transmission

lines with a small active power flow turns out to reduce the MW interdependency between islands, thus promoting load-generation balance. Finally, the algorithm is capable of restricting the splitting cutset only to the lines equipped with synchro-check relays, as only these lines can be reconnected during the restoration process (see the transmission line availability constraint in [5]). However, it should be noted that a practical ICI algorithm requires many additional factors to be taken into account. Including all relevant aspects would require significantly more space. Therefore, the presented ICI implementation in PowerFactory is a basic algorithm which may serve as an illustration of the concept.

The rest of the chapter is organised as follows. Section 11.2 provides a brief review of the slow coherency theory which is used to determine the coherent groups of generators (CGG). Section 11.3 explains the graph partitioning approach utilised to find the lines with a small MW power flow while excluding some unavailable branches (e.g. lines without synchro-check relays or transformers) and satisfying the generator coherency constraint. Section 11.4 gives an overview of the ICI program structure written in Python scripting language for PowerFactory and MATLAB. Section 11.5 provides a case study of the illustrated ICI method on the PST 16 benchmark system. Finally, Sect. 11.6 summarises the approach and gives some concluding remarks.

11.2 Slow Coherency

An important task in the design of an ICI algorithm is to identify the areas in a power system which should be separated from each other in case of instability. While the actual borders of the ICI areas may change depending on the loading condition, it is important to ensure that generators in each area synchronise after the network is separated in controlled manner. One approach to meet this requirement is to utilise the slow coherency theory developed in [8–10]. It has been pointed out in [9] that generators forming a slow coherent group, i.e. generators swinging together at oscillatory frequencies of slow inter-area modes, have a relatively strong dynamic coupling between each other. In other words, weak connections in a power network manifest themselves through slow coherency [9].

Therefore, it is prudent to utilise slow coherency identification approaches in order to find generators which should be grouped together for the purpose of ICI. The load buses corresponding to each CGG can be identified by using a graph partitioning algorithm like one described in Sect. 11.3.

Slow coherency identification approaches [8–10] are model-based methods suitable for offline computations. They are based on calculation of right eigenvectors of the electromechanical model of the power system (see Sect. 11.2.1) corresponding to the dominant slow modes. The motivations underlying this approach can be found in [8, 11]. It should be noted that significant changes in the power system operating condition, such as topology changes or large load steps, may cause the *weakly coherent* generators to change their CGG [12]. Therefore,

signal-based slow coherency approaches, such as [12], are preferable for a practical ICI implementation in a physical power system.

Given the above considerations, the overall slow coherency grouping approach can be summarised as follows:

- Formulate the electromechanical model of the studied EPS (see Sect. 11.2.1).
- Linearise the model and find its state-space representation (see Sect. 11.2.2).
- Identify the r slowest electromechanical modes of the linearised model (see Sect. 11.2.3).
- Compute the right eigenvectors corresponding to the r slowest modes. Combine the eigenvector columns into a matrix and group its rows based on the grouping algorithm as described in Sect. 11.2.4. As each row corresponds to a generator, the resulting grouping will reveal the CGGs.

11.2.1 Electromechanical Modelling

The well-known electromechanical model of an n -machine power system can be derived given the following assumptions [13]:

- The mechanical power input of synchronous generators is constant.
- Generator mechanical damping and asynchronous power are negligible.
- Synchronous generators can be represented in the network by the constant-emf-behind-the-transient-reactance model.
- The generator rotor angle coincides with the angle of the emf behind the transient reactance.
- Loads can be represented by constant impedances.

Given the assumptions above, the electrical network can be represented as shown in Fig. 11.1.

The network shown in Fig. 11.1 can be reduced to contain only the generator internal buses (i.e. the buses behind the transient reactances) by using the procedure called Kron reduction [13]. The main idea of this procedure is to eliminate the nodal equations of the original network which have zero current injections. As only the

Fig. 11.1 Electromechanical model of a multi-machine power system

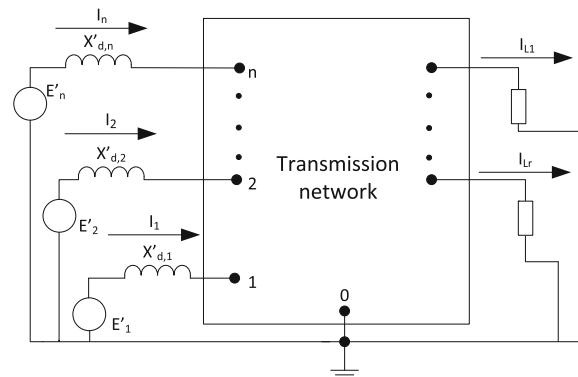
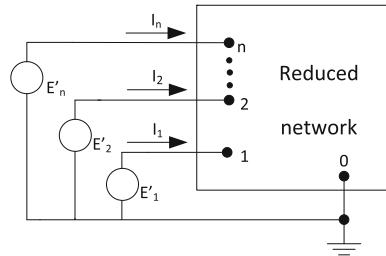


Fig. 11.2 Electromechanical model reduced to internal nodes of generators



generator internal buses' nodal equations have nonzero current injections, the voltages of the remaining network nodes can be represented as a linear combination of the internal generator voltages by exploiting the fact the left-hand sides of the network equations for the remaining nodes are zero. The reduced network obtained from the Kron reduction contains the equivalent admittances between every pair of internal generator nodes (i.e. the reduced network represents a full graph). Its graphical representation is shown in Fig. 11.2.

The reduced electromechanical model can be described by (11.1a) and (11.1b); see [9, 13].

$$\dot{\delta}_i = \omega_0(\omega_i - 1), \quad i = 1, \dots, n \quad (11.1a)$$

$$2H_i\dot{\omega}_i = - \sum_{\substack{j=1 \\ j \neq i}}^n E'_i E'_j B_{ij} \sin(\delta_i - \delta_j) - \sum_{\substack{j=1 \\ j \neq i}}^n E'_i E'_j G_{ij} \cos(\delta_i - \delta_j) - D_i(\omega_i - 1) + P_{m,i} - E'_i G_{ii}^2, \quad i = 1, \dots, n \quad (11.1b)$$

where δ_i is the rotor angle of generator i in rad, ω_i is the rotor speed of generator i per unit, H_i is the inertia constant of generator i in s, D_i is the damping coefficient of generator i per unit, $P_{m,i}$ is the mechanical power of generator i per unit, E' is the constant voltage behind the transient reactance per unit, G_{ij} and B_{ij} are the real and imaginary components of the $(i,j)^{th}$ entry of the admittance matrix of the reduced network (see Fig. 11.2) per unit and ω_0 is the base frequency in rad s⁻¹.

11.2.2 Linearised Model

The coherency behaviour of the generators can be more easily understood from the linearised electromechanical model. Equations (11.1a) and (11.1b) can be linearised about an equilibrium $\delta_i = \delta_{i,0}$ and $\omega_i = 1$, where $\delta_{i,0}$ is the equilibrium rotor angle of the i th generator. The equilibrium rotor angles of all generators can be obtained in a convenient fashion by calculating the power flow solution for the original

electromechanical model in Fig. 11.1 for the loading condition of interest. The details of this procedure in PowerFactory are given in Sect. 11.4.

The resulting linearised model derived from (11.1a) and (11.1b) is described by (11.2a) and (11.2b); see [8, 9].

$$\Delta\dot{\delta}_i = \omega_0\Delta\omega_i, \quad i = 1, \dots, n \quad (11.2a)$$

$$2H_i\Delta\dot{\omega}_i = -D_i\Delta\omega_i - \sum_{j=1}^n k_{ij}\Delta\delta_j, \quad i = 1, \dots, n \quad (11.2b)$$

where $\Delta\delta_i = \delta_i - \delta_{i,0}$, $\Delta\omega_i = \omega_i - 1$ are the small perturbations of the rotor angles and speeds around their equilibrium values and the terms k_{ij} are according to (11.3a) and (11.3b).

$$k_{ij} = -E'_i E'_j [B_{ij} \cos(\delta_{i,0} - \delta_{j,0}) - G_{ij} \sin(\delta_{i,0} - \delta_{j,0})], \quad j \neq i \quad (11.3a)$$

$$k_{ii} = -\sum_{\substack{j=1 \\ j \neq i}}^n k_{ij} \quad (11.3b)$$

Equations (11.2a) and (11.2b) can be written in the matrix form as (11.4).

$$\begin{bmatrix} \Delta\dot{\delta}_1 \\ \Delta\dot{\delta}_2 \\ \vdots \\ \Delta\dot{\delta}_n \\ \Delta\dot{\omega}_1 \\ \Delta\dot{\omega}_2 \\ \vdots \\ \Delta\dot{\omega}_n \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \omega_0 & 0 & \cdots & 0 \\ & 0 & \omega_0 & \cdots & 0 \\ & & \vdots & \vdots & \vdots \\ & & 0 & 0 & \cdots & \omega_0 \\ -\frac{1}{2}\mathbf{H}^{-1}\mathbf{K} & & -\frac{1}{2}\mathbf{H}^{-1}\mathbf{D} & & & \end{bmatrix} \begin{bmatrix} \Delta\delta_1 \\ \Delta\delta_2 \\ \vdots \\ \Delta\delta_n \\ \Delta\omega_1 \\ \Delta\omega_2 \\ \vdots \\ \Delta\omega_n \end{bmatrix} \quad (11.4)$$

where

$$\mathbf{H} = \text{diag}(H_1, H_2, \dots, H_n)$$

$$\mathbf{D} = \text{diag}(D_1, D_2, \dots, D_n)$$

$$\mathbf{K} = [k_{ij}]$$

By expressing $\Delta\omega_i = \frac{\Delta\dot{\omega}_i}{\omega_0}$ and neglecting damping, it is possible to reduce (11.4) and (11.5), which is the common form of the power system electromechanical model used for slow coherency analysis [8, 9]. The motivation to represent synchronous generators by simplified 2nd order models and to neglect damping is based on the observation that the CGGs do not depend significantly on the level of detail used in modelling the generating units [8, 11].

$$\Delta \ddot{\delta} = -\frac{1}{2} \mathbf{H}^{-1} \omega_0 \mathbf{K} \Delta \delta \quad (11.5)$$

The properties of the linearised dynamic models (11.4) and (11.5) can be analysed by studying the eigenvalues and eigenvectors of their state matrices. In particular, the standard small-signal stability analysis [14, 15] can be performed for the model described by (11.4) in order to extract mode shapes corresponding to the electromechanical state variables $\Delta\delta$ and $\Delta\omega$. It should be noted that Eq. (11.5) fully describes the properties of the complete state-space model (11.4), given that the damping is neglected in (11.4). In particular, if λ_i is an eigenvalue of the state matrix of (11.5), then $\pm\sqrt{\lambda_i}$ are the eigenvalues of the state matrix of (11.4) with all damping constants set to zero [9].

Despite the fact that the model (11.5) is commonly used in the literature to analyse slow coherency, the standard state-space model (11.4) is better suitable for the slow coherency analysis with PowerFactory, as it is the model that is readily available through the Modal Analysis Toolbox of PowerFactory.

As the complete state-space model (11.4) is being utilised, it may be useful to review the difference between the terms *mode* and *eigenvalue*. A real eigenvalue corresponds to a non-oscillatory mode, while a complex conjugate eigenvalue *pair* corresponds to an oscillatory mode in the time-domain response of the linearised power system model [14].

11.2.3 Selection of Number of Slowest Modes

Slow coherency is defined as coherency with respect to the slowest modes of a system [9]. Although there are algorithms available to group generators in a power system with respect to the r slowest modes, where r is a predefined integer, the optimal value of r is not always known. Several references, e.g. [9], use the *eigengap* heuristic (11.6) in order to determine the point of separation between slow and fast electromechanical modes.

$$\varepsilon_i = |\text{Im}(\lambda_{k+2})| - |\text{Im}(\lambda_k)|, \quad k = 3, 5, \dots, 2n - 3, \quad i = 1, 2, \dots, n - 1 \quad (11.6)$$

where λ_k is the k th complex conjugate eigenvalue of the state matrix of Eq. (11.4), and all λ_k have been sorted in the increasing order of their imaginary parts. Given such ordering of eigenvalues, their counting starts at 3, because the state matrix of (11.4) has one zero eigenvalue and one small negative eigenvalue as the only real eigenvalues. All further eigenvalues come in complex conjugate pairs, and only the eigengaps between the slowest oscillatory modes are of interest.

Given Eq. (11.6), the number of slowest electromechanical modes to be considered for generator grouping can be expressed as follows:

$$r = \arg \max_i \varepsilon_i + 1 \quad (11.7)$$

where the increment of one in (11.7) is necessary and can be related to the two real eigenvalues of the state matrix in (11.4) which correspond to the common motion of the rotor angles and speeds of all generators [9]. In other words, the minimal number of slowest modes to separate the network is two, whereby the minimal value of $\arg \max_i \varepsilon_i$ is one.

11.2.4 Generator Grouping Algorithm

The r right eigenvectors corresponding to the r slowest modes of the power system model (11.4) serve as an input for the coherency grouping algorithm. As pairs of complex conjugate eigenvalues correspond to one oscillation mode, it is only necessary to take eigenvectors corresponding to one of two complex conjugate eigenvalues. It is possible to use both eigenvector entries related to rotor angles and to rotor speeds (i.e. both rotor angle and rotor speed mode shapes), as they show the same pattern. The rotor speed eigenvector entries are finally adopted for the analysis.

Among several slow coherency identification algorithms available in the literature, the so-called tight coherency grouping algorithm [10] is chosen to find CGGs for the purpose of ICI. It is capable of automatic detection of the number of CGGs. Moreover, CGGs identified by using this algorithm usually consist of generators which are electrically close. The generator grouping procedure [10] has several presteps that are given below.

- Combine the rotor speed mode shapes obtained from the r slowest eigenvectors into a matrix \mathbf{V}_s consisting of n rows and r columns.
- Normalise the columns of the matrix \mathbf{V}_s to have the length one.
- Define the slow coherency similarity between machines i and j as the cosine of the angle between w_i and w_j , which are the respective rows of the matrix \mathbf{V}_s .

$$d_{ij} = \frac{w_i w_j^T}{\|w_i\| \|w_j\|} \quad (11.8)$$

where $\|\cdot\|$ represents the vector length, i.e. the Euclidean norm of a vector.

- Define a tolerance γ usually ranging from 0.9 to 0.95. If the slow coherency similarity (11.8) between machines i and j is larger than γ , the machines are said to be coherent.

- Define a coherency matrix \mathbf{C} as

$$[C_{ij}] = d_{ij} - \gamma \quad (11.9)$$

Only the extraction of *loose* coherent areas from the algorithm [10] is described below and implemented in MATLAB. The complete algorithm is available in the MATLAB-based Power System Toolbox (PST) [16], which is available online. Based on the given presteps, the set of rules to decide on *loosely* coherent generator groups can be summarised as follows.

- Machines i and j are coherent if $[C_{ij}] > 0$.
- If machines i and j are coherent and machines j and k are coherent, then machines i and k are also coherent.

It was observed empirically that the above algorithm usually performs well at identifying generator slow coherency. Higher values of the tolerance parameter γ correspond to smaller and tighter CGGs which tend to be robust even to significant changes in the power network (e.g. topological changes). Therefore, the value of γ of 0.95 is assumed for the coherency estimation in the subsequent sections.

11.3 Graph Partitioning

The slow coherency method presented in Sect. 11.2 solves the problem of identifying which generators can be grouped together for the purpose of ICI. For the complete islanding solution, a set of lines to be opened needs to be determined based on the multiple constraints or a subset of constraints (see Sect. 11.1).

The splitting cutset determination procedure is based on graph partitioning and aims at identifying the lines carrying the least amount of active power flow (shortly referred to as MW-flow). Its main steps are summarised below:

- Construct a weighted undirected graph $G = (V, E, W)$ representing the MW-flows in an electric power network consisting of m buses. The nodes and edges of G can be denoted as $v_i \in V$, $i = 1, 2, \dots, m$ and $e_{ij} \in E \subset V \times V$, $i = 1, 2, \dots, m$, respectively. The weight $w_{ij} = W(e_{ij})$, $i = 1, 2, \dots, m$ of the edge e_{ij} represents the averaged active power flow through the respective power network branch.
- Reduce the graph G by following the guidelines presented in Sect. 11.3.1, which are largely based on [17] and [18]. The graph reduction procedure primarily serves the purpose of incorporation of generator coherency and transmission line availability constraints into the graph partitioning, but it also increases the computational efficiency by reducing the size of the problem.
- Apply the spectral clustering method briefly described in Sect. 11.3.2 to the reduced graph.

- Post-process the output of spectral clustering as illustrated in Sect. 11.3.3 in order to identify the resulting islands in the power network.

11.3.1 Graph Reduction

The MW-flow graph G can be reduced in two steps by following the corresponding steps of the procedure outlined in [17]:

- Collapse the edges of G corresponding to network elements which cannot be included into an islanding cutset (e.g. transformers and lines without synchro-check relays; see [5]) to single nodes. Such graph edges are further referred to as *unavailable edges*, as the corresponding power network branches are referred to as *unavailable network branches* [5].
- In the graph obtained after the reduction of unavailable edges, find subnetworks corresponding to the previously identified CGGs, e.g. using a shortest path algorithm as in [17]. Merge the found subnetworks into single nodes. Obtain the connectivity and weights of the final reduced graph (further referred to as G^R) by following the guidelines presented in [17, 18]. The number of nodes in G^R is m_R .

All cuts of the final reduced graph inherently satisfy the generator coherency and transmission line availability constraints. It is worth to note that, depending on the utilised subnetwork construction algorithm, the search for the CGG subnetworks may require multiple initialisations. However, as the utilised coherency algorithm is an offline model-based technique, the subnetworks do not need to be produced in an online fashion. In other words, the graph reduction is essentially an offline procedure; see [17].

11.3.2 Spectral Clustering

Spectral clustering is a well-established clustering technique based on graph representation of the input dataset [19]. Since electric power networks can be naturally represented as graphs, it is appealing to use spectral clustering for the identification of power network buses which are tightly coupled in terms of active power flow [5, 20]. In order to produce islands of balanced size, the *normalised* graph Laplacian matrix \mathbf{L}_n is preferable for spectral clustering [19, 20]. Given the aforementioned definition of graph G , it can be computed according to (11.10); see [5, 21].

$$[\mathbf{L}_{nij}] = \begin{cases} 1, & \text{if } i = j \\ \frac{-w_{ij}}{\sqrt{d_i}\sqrt{d_j}}, & \text{if } i \neq j \text{ and } (i, j) \in E \\ 0, & \text{otherwise} \end{cases} \quad (11.10)$$

where $d_i = \sum_{j=1}^m w_{ij}$ is the weighted degree of the node v_i . For the power flow graphs other than G , the definition (11.10) should be adjusted accordingly. The normalised Laplacian of the reduced graph G^R described in Sect. 11.3.1 is of interest for the purpose of ICI. It is further referred to as $\mathbf{L}_{n,R}$.

With having $\mathbf{L}_{n,R}$ computed, the next step is to calculate its first r smallest eigenvalues and the corresponding eigenvectors, where r previously denoted the number of CGGs in the EPS; see Sect. 11.2. The number of computed eigenvectors usually corresponds to the desired number of islands [19]. That is, the goal is to identify a separate island for each CGG. The computed eigenvectors are combined into the matrix $\mathbf{X} = \mathbb{R}^{m_R \times r}$. According to [19], each row of the matrix \mathbf{X} should be normalised to have length 1. The row normalisation process results in the matrix $\mathbf{Y} = \mathbb{R}^{m_R \times r}$, the rows of which represent the m_R coordinates of points in the r -dimensional Euclidian space. This so-called *spectral r-embedding* [5] reveals the clustering structure of the m_R reduced power network nodes with respect to active power flows between them.

11.3.3 Identification of Islands

Spectral embedding does not take the actual interconnections between the nodes in the input graph into account. In order to overcome this issue, it was recommended in [21] to define a new metric in spectral embedding which measures the distances between the points according to their connectivity in the underlying graph G^R . This is essentially equivalent to the creation of a new graph G_{SC}^R which has the same sets of nodes and edges as G^R , but the edge weights are redefined according to the Euclidean distances between the respective points in the spectral embedding. Then, the distance between any two points in the spectral embedding is defined as the shortest path distance between the respective nodes of G_{SC}^R .

As each show-coherent generator group is represented by a single node in G^R and in the resulting spectral embedding, it is possible to determine the boundaries of the islands by assigning the remaining load buses to the nearest (in the sense of the distance metric defined above) CGG. In this way, each identified CGG becomes assigned to its own partition.

After clusters of nodes have been identified by following the above procedure, the nodes of each cluster are mapped back to the nodes of the original graph, and the potential cutset is defined as the edges between the buses belonging to different clusters. By using the identified cutset, it is possible to separate the CGG which goes out-of-step with the rest of the network.

11.4 ICI Program Implementation

The PowerFactory implementation of the presented simplified ICI algorithm is based on the coupling of the DIgSILENT PowerFactory [22] and MATLAB [23] software tools through Python. In this way, the advantages of both software environments can be combined in order to implement more sophisticated algorithms. The mentioned coupling has become possible since releases 15.1 of PowerFactory and 8.4 (R2014b) of MATLAB. Release 15.1 of PowerFactory has introduced a dynamic Python module `powerfactory.pyd` as a means to interface PowerFactory with Python. Release 8.4 of MATLAB has introduced MATLAB Engine for Python, which allows to start or connect to MATLAB from Python.

Python is a non-proprietary high-level general-purpose interpreted programming language, which supports both object-oriented (OOP) and procedural programming paradigms. Python has been introduced to PowerFactory as an alternative to the built-in DIgSILENT programming language (DPL). The *PowerFactory* Python module provides the equivalents for the majority of functionalities available via DPL. By importing the *PowerFactory* Python module inside of a Python script, it is possible to control PowerFactory from the Python environment in the same way as it is possible with DPL. Moreover, the rich programming capabilities of Python become available for processing of data obtained from PowerFactory. Lastly, it becomes relatively easy to send data obtained from PowerFactory to other applications which also have an interface with Python (e.g. to MATLAB) and to receive data back from those applications to PowerFactory. An important additional advantage of Python scripting over DPL is the possibility to use debugger tools included into many Python Integrated Development Environments (IDEs).

In order to control PowerFactory via Python, a Python script file needs to be created externally on the hard drive and linked to PowerFactory via a `*.ComPython` object. An important difference between `*.ComDPL` objects for DPL scripts and `*.ComPython` objects for Python scripts is that the actual script is not stored inside of the latter ones. It is also possible to control PowerFactory via Python without creating a `*.ComPython` object. This can be accomplished by running PowerFactory in engine mode (see the User Manual [22] for more information).

11.4.1 ICI Program Components

The capabilities of Python as a mainstream programming language facilitate more structured and sophisticated program designs for PowerFactory (as compared to DPL). In particular, the support of OOP by Python is useful for writing larger programming projects related to PowerFactory.

The ICI program is a small-scale project written in Python and MATLAB languages. Its components are designed with the idea of code modularity in mind,

which is done in order to promote the code reuse. Although the program codes are too extensive to put them inside of the chapter, it is still possible to describe the functionalities of the main components of the project.

`ICI_PST16.py`: This file contains the top-level Python script which is started from PowerFactory and calls all other Python and MATLAB functions and class methods.

`pypf.py`: This file contains a Python class whose attributes link to the relevant data of the investigated PowerFactory project (e.g. to all network branches, terminals and generators). The class also contains several methods to manipulate the data contained in its attributes (`rename_buses_branches`, `extract_flows`, `insert_ici_events`, `extract_eig`) as well as auxiliary methods to maintain the class contents.

`pyUtils.py`: This file contains a Python module with auxiliary functions that are used inside of the methods of the `pypf` class.

`formEigVecMatr.m`: This MATLAB function selects the r eigenvectors corresponding to the r slowest system modes and returns the matrix \mathbf{V}_s introduced in Sect. 11.2.4. The number r can be either predefined or estimated based on (11.6) and (11.7).

`coh_loose.m`: This MATLAB function implements the generator coherency grouping algorithm described in Sect. 11.2.4.

`digs12graph.m`: This MATLAB function converts branch power flows extracted from PowerFactory to the MATLAB representation of the power flow graph G from Sect. 11.3. The graph G is modelled in MATLAB by its adjacency and incidence matrices. The function can also convert a list of unavailable edges extracted from PowerFactory to a MATLAB-compatible representation.

`COSC.m`: This MATLAB function implements the constrained graph partitioning algorithm described in Sect. 11.3 and returns the resulting cutset to Python. In order to accomplish this, it makes use of `digs12graph.m` as well as about 10 other dedicated MATLAB functions.

11.4.2 Interaction Between Program Components

As the program is comprised of pieces which are largely independent, it is useful to illustrate the capabilities and behaviour of the separate components on an example that involves their interaction. The top-level Python script used for the simulation of the ICI case study in Sect. 11.5 is chosen as such an example. Its flow chart is depicted in Fig. 11.3. Although the original Python script implements a particular ICI test case, its flow chart in Fig. 11.3 is more generic and may correspond to a variety of system instability scenarios followed by ICI. Some comments about the flow chart steps are given below.

1. A link between the electric network model in PowerFactory and the graph G defined in Sect. 11.3 needs to be established in order to map the ICI solutions

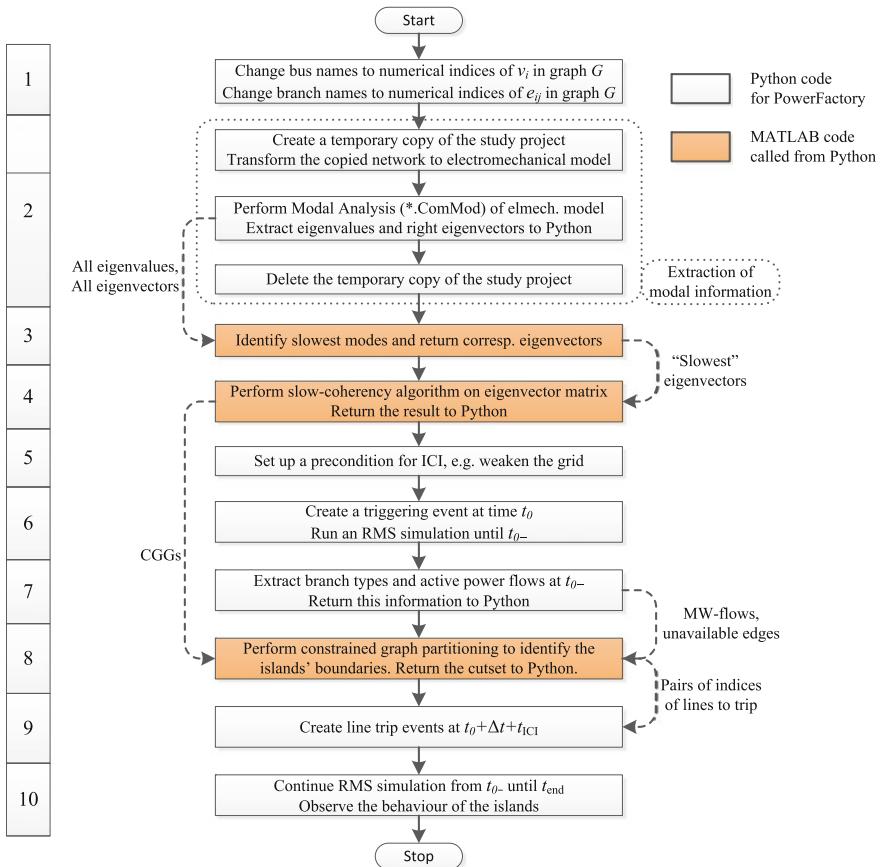


Fig. 11.3 Top-level ICI program structure

obtained for the graph G back to the network. As the renaming step is irreversible, it is recommended for the purpose of ICI study to create a separate copy of the investigated project.

- Slow coherency is identified based on the electromechanical model of power system, which implies 2nd order generator models with zero damping and all generator controls disabled. As actual power networks are rarely modelled with such assumptions, it is convenient to make a temporary copy of the active project and to modify it accordingly. Then, the output of the modal analysis command (*.ComMod) performed on the modified project copy is returned back to Python. This output is comprised of all eigenvalues and right eigenvectors of the electromechanical model (11.4). Finally, the temporary copy of the project is deleted.

3. The r right eigenvectors corresponding to the r slowest modes are returned combined to the matrix \mathbf{V}_s . The number r can be either predefined or identified based on (11.6) and (11.7).
4. This step implements the identification of *loosely* coherent generator groups following the description in Sect. 11.2.4.
5. Usually, some sequence of adverse events precedes a network instability that causes ICI to operate. This sequence of events can often be reproduced by a script that changes the default network condition accordingly and, if necessary, rolls it back upon completion of the main program.
6. For a given instability precondition, a simulation event needs to be created that actually triggers the instability during the time-domain simulation run. The time instant of this event is denoted as t_0 . After the time Δt following the triggering event, the instability is detected, which initiates the execution of the ICI algorithm.
7. Programmatically, it is easier to extract power flows in the network at t_{0-} (i.e. nearly at the time of the triggering event, but not including the triggering event) by first running the RMS-type time-domain simulation until t_{0-} and then extracting the resulting power flow variables for each network branch after the simulation has stopped.
8. This step implements the constrained graph partitioning algorithm outlined in Sect. 11.3.
9. This step inserts transmission line trip events which are used to separate the network after the RMS-type time-domain simulation resumes at Step 10. The time t_{ICI} represents an additional time delay related to the calculation of the islanding cutset and the actual network separation.
10. Resume the RMS simulation at t_{0-} in order to simulate the network separation and the resulting post-islanding transients.

In the algorithm flow chart in Fig. 11.3, slow coherent generator groups for the default network configuration are taken as input for islanding. Although it makes the algorithm less adaptive to the actual operating condition, the offline calculation and analysis of CGGs are easier to implement. This assumption allows to keep the ICI implementation at a manageable level. The assumption can be justified by the fact that slow coherent generator groups represent a fundamental property of the network related to its topological structure [9, 11] and usually do not experience significant changes.

11.5 ICI Case Study

This section presents simulation results for the sample ICI algorithm. The high-level structure of the case study has already been mentioned in Sect. 11.4 in the form of a flow chart. This section presents a particular scenario, in which the sample ICI algorithm is applied to mitigate a wide-area instability in the PST 16 benchmark system.

11.5.1 Test System

The ICI case study is based on the PST 16 benchmark system [24]. It consists of three meshed areas, 66 buses, 16 generators, 28 transformers and 51 transmission lines. Due to the unbalanced load and generation in the areas and the presence of long inter-area tie lines, the PST 16 test system is useful for studies of various stability problems [24]. The slightly modified version of the PST 16 test system model is used for the simulations. In particular, the thyristor-controlled series compensator (TCSC) has been removed, and the tie line between areas A and C is modelled to have a half of its original impedance (i.e. as a double-circuit line). The test system is shown in Fig. 11.4 together with the final network separation result.

In the nominal operating condition, all network elements are in service. The loads were slightly adjusted in order to produce a more distinct and realistic sequence of events leading to instability. The resulting active load and generation for each area are summarised in Table 11.1.

As it can be seen, Area A has a significant excess of generation, and it, in fact, supplies power to the neighbouring areas. At the same time, Area C has a significant excess of load, while Area B is more balanced. Area C is the main power consumer, and it is largely supplied from Area A, which leads to a significant power flow through tie line A–C. This power flow may become especially large if tie line A–B is disconnected, and power cannot be sent from Area A to Area B directly (cf. Sect. 11.5.2). Thus, the separation of the areas according to the tie lines can lead to a poor power balance and large changes in electric power outputs of the generators.

11.5.2 Wide-Area Instability Scenario

The PST 16 test system represents a relatively small power network with a limited number of realistic scenarios of a wide-area instability. The following instability scenario has been finally chosen:

- Transmission line C4–C6 is out of service due to maintenance.
- Tie line A–B trips due to a sustained short-circuit.
- As the consequence of the power flow redistribution, transformer station C8 3T is disconnected due to a 75% overload.
- The disconnection of transformer C8 3T causes wide-area rotor angle instability accompanied by low voltages in Area C.

Controlled islanding should be applied after the disconnection of transformer C8 3T as soon as the instability has been detected. Without loss of generality, an advanced instability detection mechanism capable of identifying the out-of-step condition before the angular differences reach 180° is assumed for this case study. However, as a controlled islanding algorithm may in practice require an additional

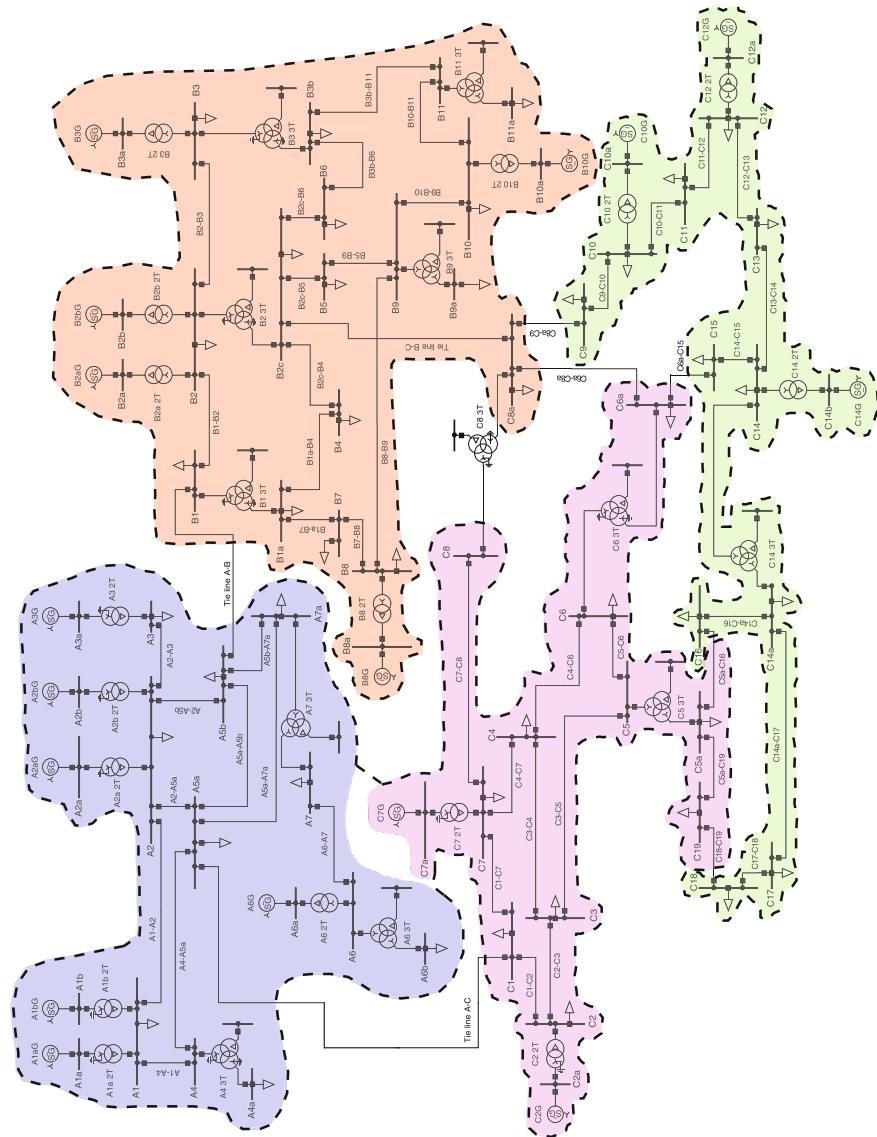


Fig. 11.4 PST 16 test system [24]. Blue: area of CGG-1; magenta: area of CGG-2; green: area of CGG-3; red: area of CGG-4. The areas have been determined by using the graph partitioning algorithm. The boundaries of the resulting islands are shown with dashed lines

Table 11.1 Load and generation in PST 16 system

	Active load (MW)	Generation (MW)
Area A	2535	5082
Area B	7215	6627
Area C	7833	6051
Total	17,583	17,761

non-negligible amount of time to calculate a solution, it is desirable to predict the emerging instability straight after the triggering event.

11.5.3 Controlled Islanding

The first step of the ICI algorithm is to perform slow coherency analysis of the PST 16 network for the nominal operating condition. The eigenvalues and right eigenvectors of the electromechanical model (11.4) are obtained with the built-in Modal Analysis command of PowerFactory. The 16 smallest eigenvalues are $\{0, -0.0548, \pm 3.142j, \pm 3.718j, \pm 5.021j, \pm 5.265j, \pm 5.716j, \pm 5.894j, \pm 6.099j\}$. As it can be seen, there is a large gap between the complex eigenvalues $\pm 3.718j$ and $\pm 5.021j$. Therefore, the number of eigenvectors for slow coherency analysis is three [8–10], and the generator grouping algorithm of Sect. 11.2.4 is performed with the eigenvectors corresponding to the eigenvalues $\{0, 3.142j, 3.718j\}$. The resulting grouping for $\gamma = 0.95$ is presented in Table 11.2.

As it can be seen, four coherent generator groups have been identified. The tolerance value γ for the coherency grouping algorithm has been taken at the highest recommended value of 0.95, which has resulted in four identified coherent groups. It is worth noting that the groups would be the same for γ equal to 0.9, and in general the CGGs in Table 11.2 are tight. The boundaries of each coherent generator group in terms of predisturbance active power flow are determined by the graph partitioning algorithm described in Sect. 11.3 and represented in Fig. 11.4.

After the disconnection of transformer C8 3T, the resulting unstable transient is shown in Fig. 11.5. As it can be seen, the actual out-of-step condition involves the loss of synchronism between CGG-3 and the rest of the network. Therefore, the

Table 11.2 Slow coherent groups of generators in the PST-16 network

Generator group	Generator buses
CGG-1	A1aG, A1bG, A2aG, A2bG, A3G, A6G
CGG-2	C2G, C7G
CGG-3	C10G, C12G, C14G
CGG-4	B2aG, B2bG, B3G, B10G, B8G

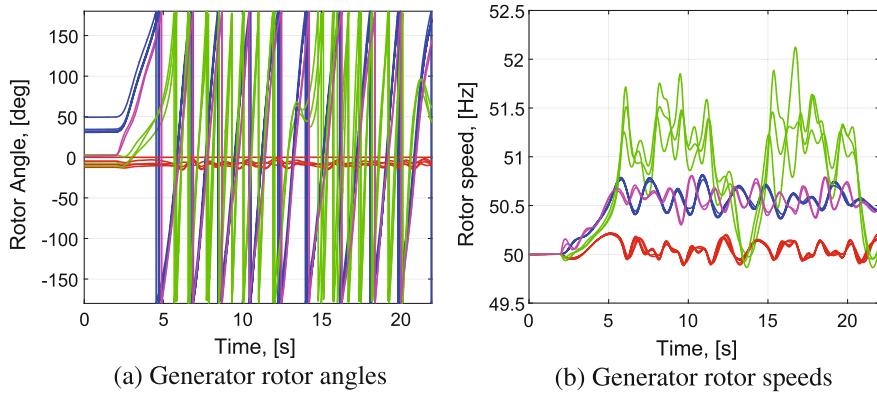


Fig. 11.5 Time-domain simulation of wide-area instability. Blue: signals of CGG-1; magenta: signals of CGG-2; green: signals of CGG-3; red: signals of CGG-4

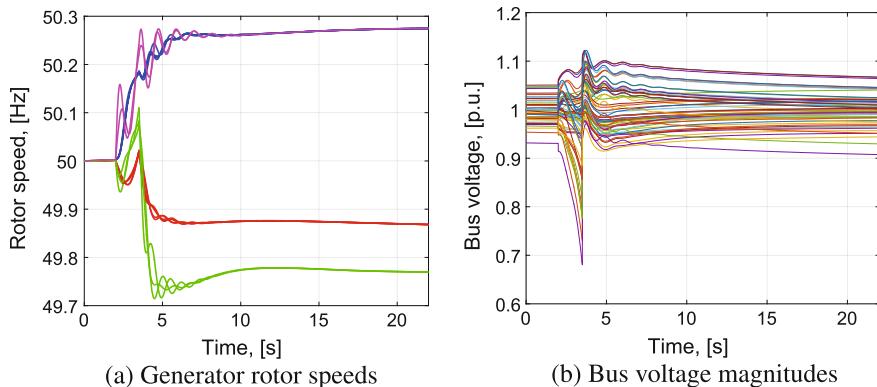


Fig. 11.6 Time-domain simulation of ICI

cutset between the area of CGG-3 and the rest of the network needs to be tripped. The resulting transient responses can be observed in Fig. 11.6. The time period Δt from the disconnection of transformer C8 3T to the initiation of the controlled islanding algorithm is 1 s. The time period t_{ICI} to compute and implement islanding is assumed 0.5 s, which is around the expected time for a controlled islanding algorithm to return a solution [5].

The resulting islands have a good power balance with their steady-state frequencies close to the nominal frequency of 50 Hz and the maximal voltage deviation not exceeding 0.1 p.u. In certain cases, the initial splitting boundary may also require a post-processing in order to improve the power balance of the islands. However, the multiple additional questions arising in the design of adaptive power network separation schemes are beyond the scope of the chapter.

11.6 Conclusion

A simplified ICI procedure to adaptively separate the network following a wide-area instability has been presented in this chapter. The procedure focuses on grouping the coherent generators together and finding the minimal active power flow cut. The methodology is applied to the PST 16 benchmark system in order to show the feasibility of the approach. The algorithm has been implemented by using the Python scripting language, which is available in PowerFactory since version 15.1. Through Python, PowerFactory could gain access to MATLAB and call the MATLAB components of the ICI code. Moreover, the Python part of the ICI code takes advantage of the object-oriented capabilities of the Python programming language.

It has been mentioned throughout this chapter that the original ICI problem is a very comprehensive one and includes many additional questions. Among others, the following issues are not considered: adaptive generator coherency estimation, voltage stability, equipment thermal limits. However, the demonstrated ICI methodology in PowerFactory can serve as the basis for the development of more advanced controlled network separation algorithms. Many of the unaddressed issues are currently being pursued as future work.

Acknowledgements This study was financially supported by the Dutch Scientific Council NWO-STW, under the project 408-13-025 within the program of Uncertainty Reduction of Sustainable Energy Systems (URSES) in collaboration with TenneT TSO and the Dutch National Metrology Institute, van Swinden laboratory.

References

1. G. Andersson, P. Donalek, R. Farmer, N. Hatziargyriou, I. Kamwa, P. Kundur, N. Martins, J. Paserba, P. Pourbeik, J. Sanchez-Gasca, R. Schulz, A. Stankovic, C. Taylor, V. Vittal, Causes of the 2003 major grid blackouts in North America and Europe, and recommended means to improve system dynamic performance. *IEEE Trans. Power Syst.* **20**(4), 1922–1928 (2005)
2. J. Romero, Blackouts illuminate India’s power problems. *IEEE Spectr.* **49**(10), 11–12 (2012)
3. V. Madani, D. Novosel, S. Horowitz, M. Adamak, J. Amantegui, D. Karlsson, S. Imai, A. Apostolov, IEEE psrc report on global industry experiences with system integrity protection schemes (sips). *IEEE Trans. Power Deliv.* **25**(4), 2143–2155 (2010)
4. B. Yang, V. Vital, G.T. Heydt, Slow-coherency-based controlled islanding-a demonstration of the approach on the August 14, 2003 blackout scenario. *IEEE Trans. Power Syst.* **21**(4), 1840–1847 (2006)
5. J. Quirós-Tortós, R. Sánchez-García, J. Brodzki, J. Bialek, V. Terzija, Constrained spectral clustering-based methodology for intentional controlled islanding of large-scale power systems. *IET Gener. Transm. Distrib.* **9**(1), 31–42 (2015)
6. Q. Zhao, K. Sun, D.Z. Zheng, J. Ma, Q. Lu, A study of system splitting strategies for island operation of power system: a two-phase method based on obdds. *IEEE Trans. Power Syst.* **18**(4), 1556–1565 (2003)

7. H. Shao, S. Norris, Z. Lin, J. Bialek, Determination of when to Island by Analysing Dynamic Characteristics in Cascading Outages, in *2013 IEEE Grenoble PowerTech* (2013), pp. 1–6
8. B. Avramovic, P.V. Kokotovic, J.R. Winkelman, J.H. Chow, Area decomposition for electromechanical models of power systems. *Automatica* **16**(6), 637–648 (1980)
9. J.H. Chow, *Time-Scale Modeling of Dynamic Networks with Applications to Power Systems, Lecture Notes in Control and Information Sciences*, vol 46 (Springer-Verlag, Berlin and New York, 1982)
10. J.H. Chow, New Algorithms for Slow Coherency Aggregation of Large Power Systems, in *Systems and Control Theory for Power Systems, IMA Volumes in Mathematics and its Applications*, vol. 64, ed. by J.H. Chow, P.V. Kokotović, R.J. Thomas (Springer-Verlag, New York, 1995)
11. H. You, V. Vittal, X. Wang, Slow coherency-based islanding. *IEEE Trans. Power Syst.* **19**(1), 483–491 (2004)
12. M.A.M. Ariff, B.C. Pal, Coherency identification in interconnected power system-an independent component analysis approach. *IEEE Trans. Power Syst.* **28**(2), 1747–1755 (2013)
13. P.M. Anderson, A.A. Fouad, *Power System Control and Stability*, 2nd edn. (IEEE Press Power Engineering Series, IEEE Press and Wiley-Interscience, Piscataway, N.J, 2003)
14. P. Kundur, N.J. Balu, M.G. Lauby, *Power System Stability and Control* (The EPRI power system engineering series, McGraw-Hill, New York, 1994)
15. G. Rogers, *Power System Oscillations The Springer International Series in Engineering and Computer Science, Power Electronics and Power Systems* (Springer, US, Boston, MA, 2000)
16. C. Jh, C. Kw, A toolbox for power system dynamics and control engineering education and research. *IEEE Trans. Power Syst.* **7**(4), 1559–1564 (1992)
17. G. Xu, V. Vittal, Slow coherency based cutset determination algorithm for large power systems. *IEEE Trans. Power Syst.* **25**(2), 877–884 (2010)
18. S. Rangapuram, M. Hein, Constrained 1-Spectral Clustering, in *International Conference on Artificial Intelligence and Statistics (AISTATS)* (2012)
19. U. von Luxburg, A tutorial on spectral clustering. *Stat. Comput.* **17**(4), 395–416 (2007)
20. L. Ding, F.M. Gonzalez-Longatt, P. Wall, V. Terzija, Two-step spectral clustering controlled islanding algorithm. *IEEE Trans. Power Syst.* **28**(1), 75–84 (2013)
21. R.J. Sanchez-Garcia, M. Fennelly, S. Norris, N. Wright, G. Niblo, J. Brodzki, J.W. Bialek, Hierarchical spectral clustering of power grids. *IEEE Trans. Power Syst.* **29**(5), 2229–2237 (2014)
22. DIgSILENT GmbH, *PowerFactory v15.2.5 User Manual* (Gomaringen, Germany, 2015) online Edition. Available at <http://www.digsilent.de>
23. MATLAB, version 8.6.0 (R2015b) (The MathWorks Inc., Natick, Massachusetts, 2015). Available at <http://www.mathworks.com> (Online)
24. S.P. Teeuwsen, *Oscillatory Stability Assessment of Power Systems Using Computational Intelligence* (Ph.d. dissertation, University Duisburg-Essen, 2005)

Chapter 12

Peer-to-Peer (P2P) MATLAB–PowerFactory Communication: Optimal Placement and Setting of Power System Stabilizer



Andrei Stativă and Francisco Gonzalez-Longatt

Abstract DIgSILENT PowerFactory is an extremely powerful power system analysis software used for simulating the electrical power networks. The smart grid envisions a modernised electrical grid that uses communication to gather and act on information. The ever-increasing communication and controls in power networks increase the complexity of the system. Co-simulation becomes essential to couple system simulators from different domains. This chapter presents the Peer-to-peer MATLAB–PowerFactory communication. The method is extremely simple file sharing approach to couple MATLAB and PowerFactory, and it is used to solve an optimisation problem. An illustrative two area power system is modelled using PowerFactory and an optimisation algorithm is implemented in MATLAB. The optimisation process is used for the optimal tuning and placement of Power System Stabilizer (PSS) in order to enhance the power system stability. The optimisation algorithm used in this chapter is an evolutionary algorithm (Particle Swarm Optimisation—PSO). MATLAB and DIgSILENT are employed and linked together in a genuine automatic data exchange procedure. Consequently, the test system and the controllers are modelled in DIgSILENT and the PSO algorithm is implemented

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_12) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

A. Stativă (✉)

Power Systems Department, “Gheorghe Asachi” Technical University of Iași, Iași, România
e-mail: stativa_andrei@yahoo.com

F. Gonzalez-Longatt

Electrical and Systems Engineering,
Loughborough University, Loughborough LE11 3TU, UK
e-mail: fglongatt@fglongatt.org

in MATLAB. For evaluating the particles evolution throughout the searching process, an eigenvalue-based multi-objective function is used. The performance of the proposed PSO-based PSS test system in damping power system oscillations is proved through eigenvalue analysis and time-domain simulations.

Keywords Modal analysis · Electromechanical oscillations · PSS
DIgSILENT · MATLAB

12.1 Introduction

Power system oscillations are complex, and they are not straightforward to analyse [1]. However, the *electromechanical oscillations* (EOs) are a common dynamic phenomenon in every power system. EOs are essentially nonlinear as observed from their rotor angle waveforms due to the nonlinear power network [2].

The EOs are due to the design of the synchronous generator, which, in order to generate electric power, uses massive rotors that respond to load perturbations by oscillating [3]. The EOs in power systems occur when the rotors of the synchronous generators from a multi-machine power system behave as rigid bodies and oscillate with respect to one another using the electrical transmission path between them to exchange energy. There are several classifications of the EOs; they can be classified by the system components that they affect. Considering the nature of electromechanical oscillations, the EOs can be classified as [4]: (i) intra-plant mode oscillations, (ii) local plant mode oscillations, (iii) inter-area mode oscillations, (iv) control mode oscillations and (v) torsional modes between rotating plant.

In the *local oscillation* mode, one synchronous generator swings against the rest of a large power system at 0.8–2.0 Hz. The *inter-area oscillation* is a phenomenon which is observed over a large part of the large power system. The inter-area mode oscillations involve two coherent groups of generators swinging against each other at 1 Hz or less (typically between 0.1 and 0.8 Hz). The variation in a tie-line power can be large [4].

The presence of the EOs has been reported and recorded since the early power systems; however, these oscillations gain more concern along with the development of massively interconnected large power systems, where the poor damping could have led to power system oscillations.

The scientific literature is rich showing studies [1, 4–6] where the lack of damping is caused due to the weak interconnection between large-scale power systems, a large amount of long-distance power transmission or negative damping of fast-acting high-gain AVR [7].

The low-frequency oscillations in a large power system create variations in the main electromechanical variables, e.g. system frequency, bus voltage or active and reactive power flows [8]. If the low-frequency oscillation is not appropriately damped, they can be sustained or even grow, leading the power system to emergency measures, including the system separation or islanding [9].

Several solutions are proposed in the scientific literature against the power oscillations issue, including the used of Flexible Alternative Current Transmission

Systems (FACTS) and Power System Stabilizers (PSSs) appropriately installed in the power systems. However, the most convenient way of damping EOs is by the use of PSS.

The PSS is recognised as a supplementary excitation controller used to damp (providing positive damping) generator EOs in order to protect the shaft line and stabilise a large power system. The PSS also damps generator rotor angle swings, which are of greater range in frequencies in power system.

From the control theory point of view, the PSS is a feedback controller, a part of the control system for a synchronous generator; it provides an additional signal that is added to the input summing point at the Automatic Voltage Regulator (AVR). The PSS generates an electric torque component through the generator excitation system proportional to the rotor speed deviation enhancing the transfer capability of the power system [10, 11, 12].

Several types of PSS are defined in the literature, but they can be classified as: speed-based stabiliser, frequency-based stabiliser, power-based stabiliser, integral of accelerating power-based stabiliser, multi-band stabiliser and others.

The PSS structure and its influence towards the system stability are explained in greater detail in [2, 11].

The most important aspects of designing a PSS are [7]: (i) the proper choosing of stabiliser's feedback signals, (ii) the optimal PSS parameter setting and the proper selection of PSS location.

In literature, the problem of the optimal setting of PSS has been intensively analysed in recent years. Although different techniques such as Pole-Placement [13], Linear Matrix Inequalities [14], Linear Quadratic Regulator Formulation [15] have been successfully used on the problem of PSS design, recently the optimisation techniques such as Tabu Search [16], Genetic Algorithms (GA) [17], Simulated Annealing [18], Particle Swarm Optimisation (PSO) [19], Mean-Variance Mapping Optimisation (MVMO) [6, 20] and others are gaining more and more attention. This is due to the fact that conventional techniques are confronted with heavy computational burden and the possibility of getting trapped in local optimum [21]. Also, the optimisation techniques have captured attention in recent years because of their simple implementation and it does not require previous problem knowledge [22].

This chapter presents the peer-to-peer (P2P) MATLAB–PowerFactory communication, which is extremely simple file sharing approach to couple MATLAB and PowerFactory and it is used to solve an optimisation problem. The chapter focuses on solving the problem of optimal tuning and placement of PSS in order to enhance the power system stability. The optimisation algorithm used in this chapter is an evolutionary algorithm (Particle Swarm Optimisation—PSO). In this chapter, the problem of optimal tuning and placement of PSSs is specifically solved for a very specific illustrative network using PSO, the rationale behind that is to allow a proper explanation of the P2P MATLAB–PowerFactory communication method.

12.2 Problem Statement: Optimal Tuning and Placement

Traditionally, the placement and tuning of PSSs has been tackled as individual problems [20] based on participation factors, residues [23], damping torque, sensitivity coefficients and singular value decomposition [24, 25].

Alternatively, the simultaneous solution for placement and tuning of PSSs has also been explored using optimisation techniques. Predominantly, the joint determination of optimal placement and coordinated tuning of PSSs (OPCPSS) constitutes a challenging optimisation problem due to the mix-integer combinatorial nature as well as to the nonlinearity, multimodality and no convexity of the search space [26].

12.2.1 Modelling Considerations

In this chapter, the problem of optimal tuning and placement of PSSs is specifically solved considering the well-known “Two-Area test system”; the test system is popularly called “Two-area Kundur test system” because the system is shown as in the example 12.6 at page 813 in the textbook “Power System Stability and Control,” written by Kundur [3]. However, the example two-area system is artificial; the model was created for a research report commissioned from Ontario Hydro by the Canadian Electrical Association [27, 28] to exhibit the different types of oscillations that occur in both large and small interconnected power systems. The single-line diagram of the system is shown in Fig. 12.1.

The test system contains eleven buses and two areas (see details in Fig. 12.1), connected by a weak tie between bus 7 and 9. Totally two loads are applied to the system at bus 7 and 9. Two shunt capacitors are also connected to bus 7 and 9 as shown in Fig. 12.1. The system has the fundamental frequency 60 Hz. The generators and their controls are identical. The system is quite heavily stressed; it has 400 MW flowing on the tie lines from Area 1 to Area 2 [1].

The test system has been modelled in DigSILENT PowerFactory 2016, including the generation unit controllers (see Fig. 12.2).

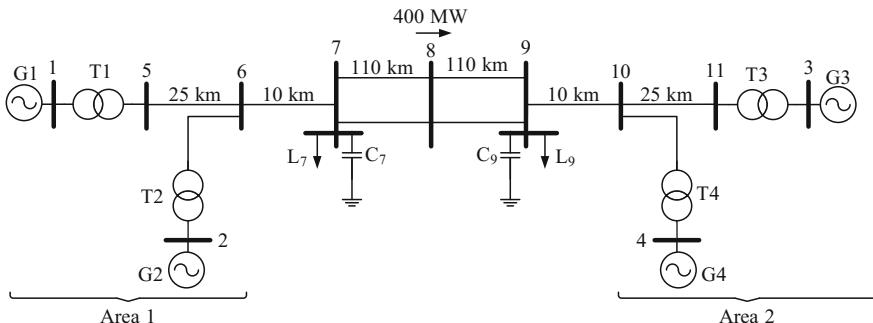


Fig. 12.1 Single-line diagram of the two-area test system [1, 3, 28]

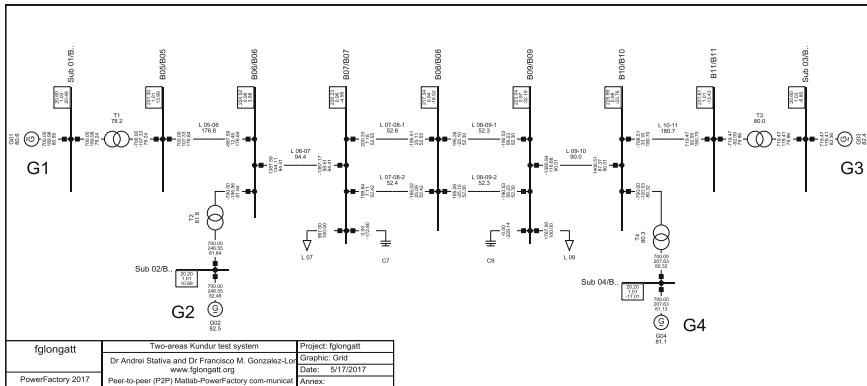


Fig. 12.2 Single-line diagram of the two-area test system implemented in DIgSILENT PowerFactory

DIgSILENT PowerFactory is equipped with a very friendly user interface which offers a simple alternative to the creation of user-defined dynamic models, *DIgSILENT Simulation Language* (DSL).

The synchronous generators are modelled in this chapter based on the well-known 6th order model represented in a rotor reference system (Park coordinates, dq -reference frame) [29]. As a consequence, the state vector (\mathbf{x}) that describes the electromechanical dynamic of the generator is described by six stated variables considering the stator and rotor flux linkage:

$$\mathbf{x} = [\omega, \phi, \phi_e, \psi_D, \psi_Q, \psi_x]^T \quad (12.1)$$

where:

- ω Speed Rotor speed [p.u.]
- ϕ phi Rotor angle [rad.]
- ϕ_e psie Excitation flux [p.u.]
- ψ_D psiD Flux in d-winding [p.u.]
- ψ_Q psiQ Flux in first quadrature axis (q-winding) [p.u.]
- ψ_x psix Flux in second quadrature axis (x-winding) [p.u.]

The generators are equipped with IEEE Type 1S excitation systems [30]. In these excitation systems, voltage (and also current in compounded systems) is transformed to an appropriate level. Rectifiers, either controlled or non-controlled, provide the necessary direct current for the generator field. The Global Library of DIgSILENT PowerFactory provides the model (avr_IEET1S). Also, the generator units are equipped with IEEE Type 2 Speed-Governing Model.

Also, the generators are equipped with a Power System Stabilizer (PSS) or damping controller; it is a speed sensitive PSS. The speed-based stabilizer is the simplest method to provide a damping torque in the synchronous machine; the controller measures the rotor speed and uses it directly as an input signal in

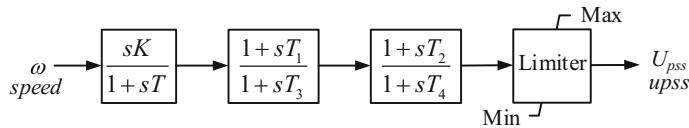


Fig. 12.3 Block diagram of the speed-based PSS controller used in this chapter

the stabiliser structure. The general structure of the speed-based PSS used in this chapter is depicted in Fig. 12.3.

The speed-based PSS uses a transfer function as follow:

$$\frac{\text{upss}}{\text{speed}} = \frac{U_{\text{pss}}}{\omega} = K \frac{s}{(1+sT)(1+sT_3)} \frac{(1+sT_1)(1+sT_2)}{(1+sT_4)} \quad (12.2)$$

where:

- K Stabiliser gain [p.u.]
- T Washout integrate time constant (s)
- T_1 First lead/lag derivative time constant (s)
- T_3 First lead/lag delay time constant (s)
- T_2 Second lead/lag derivative time constant (s)
- T_4 Second lead/lag delay time constant (s)

The speed-based PSSs are implemented in DIgSILENT PowerFactory, and the block definition and block diagram are shown in Fig. 12.4.

The proposed PSS should be integrated into the generator excitation system, and the controller parameters (gain and time constants) must be appropriately calculated in order to provide inappropriate damping of the undesired EO.

The numerical values of the washout integrate time constant (T), the second lead/lag derivative time constant (T_2) and the second lead/lag delay time constant (T_4)

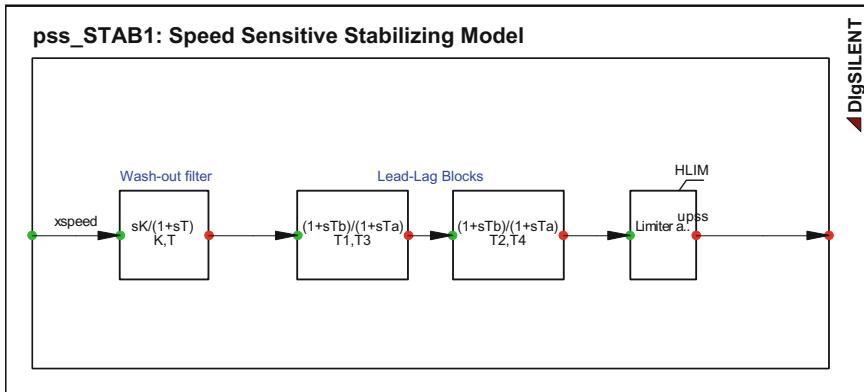


Fig. 12.4 DIgSILENT simulation language implementation of the speed-based PSS controller used in this chapter

are suggested in some publications as [21], where they assume: $T = 5.00$ s, $T_2 = 0.5$ s and $T_4 = 0.05$ s.

If the previously discussed time constant is assumed known, then the optimal setting of the PSS is conducted considering the remaining time constants as unknown. Consequently, the parameters that are considered adjustable and remain to be optimised are the PSS gain (K) and the time constants of the first lead/lag block: the first lead/lag derivative time constant (T_1) and the first lead/lag delay time constant (T_3).

An appropriate limit should be defined to the time constants in order to define a realistic search space for the optimisation method; as a consequence, the following limits are considered:

$$\begin{aligned} 1 &\leq K \leq 90 \\ 0.05 &\leq T_1 \leq 2 \\ 0.01 &\leq T_3 \leq 0.5 \end{aligned} \quad (12.3)$$

12.3 Objective Function

There are several studies that investigate the PSS action towards one critical oscillatory mode. Since trying to enhance the damping of one mode, may cause adverse effects on the other oscillatory modes.

Mathematically, the optimal placement and coordinated tuning of the power system damping controller's problem can be defined as an optimisation problem of the following format:

$$OF = \sum_{\sigma_i \geq \sigma_0} [\sigma_0 - \sigma_i]^2 + \alpha \cdot \sum_{\xi_i \leq \xi_0} [\xi_0 - \xi_i]^2 \quad (12.4)$$

where:

- σ_i The real part of the i -th eigenvalue
- ξ_i The damping ratio of the i -th eigenvalue
- α Scaling factor

The use of the previously presented objective function takes into consideration the damping of all critical modes, driving the system eigenvalues towards the desired σ_0 and ξ_0 in the left side of the s -plane.

The values of the eigenvalues are easily obtained considering the modal analysis functionality included in DIgSILENT PowerFactory. The modal analysis command (*ComMod*) calculates the eigenvalues ($\lambda_i = \sigma_i \pm j\omega_i$) and eigenvectors of a dynamic multi-machine system including all controllers and power plant models.

Herewith, the optimisation problem is to minimise OF, taking into account the constraints defined in (12.3).

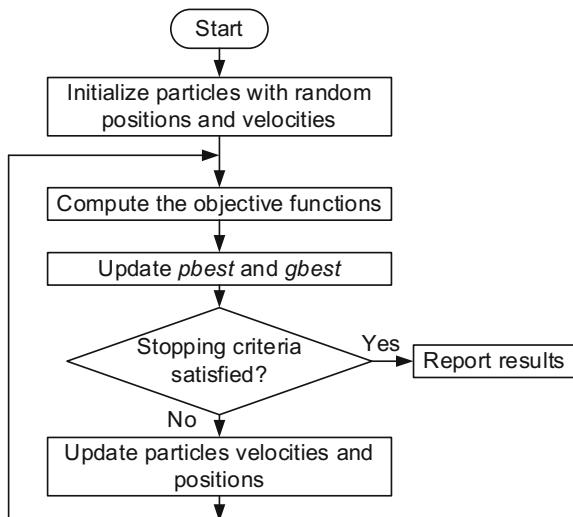
12.4 Particle Swarm Optimisation Algorithm

The particle swarm optimisation algorithm is an optimisation technique inspired by the natural movement and intelligence of bird flocks and fish schooling [2]. It was first introduced by Eberhart and Kennedy in 1995 to graphically simulate the graceful and unpredictable choreography of a swarm [31]. The main idea of the PSO algorithm consists in moving a predefined number of particles throughout the searching space to find the best solution [2]. The movement pattern of the particles towards the best solutions is defined by the social interaction between the individuals from the population.

Figure 12.3 shows a generic flow chart depicting the main steps of the PSO algorithm. Particles are used for the mathematical representation of the flock, and the particles are modelled as vectors in a multidimensional search space. The optimisation process starts by randomly generating the population and the velocities of the particles. To assign a certain measure of performance, the particles are evaluated according to an objective function. In this way, the personal best of each particle, as well as the global best of the entire population, is determined. With this information, the velocity of every individual is computed taking into account its previous velocity, personal best (pbest) and global best (gbest) (12.5). The new positions of the individuals are then updated by adding the computed velocities to the actual position according to (12.5) (Fig. 12.5).

$$\omega_i^{k+1} = W\omega_i^k + C_1 \cdot \text{rand} \cdot (\text{pbest}_i - \theta_i^k) + C_2 \cdot \text{rand} \cdot (\text{gbest} - \theta_i^k) \quad (12.5)$$

Fig. 12.5 PSO algorithm flow chart



where:

$\omega_i^{k+1} v_i^{k+1}$	Velocity of the i th particle at iteration k
W	Inertia coefficient
C_1, C_2	Weighting coefficients
p_{best_i}	Personal best of the i th particle
g_{best}	Global best of the population
θ_i^k	Position of the i th particle at iteration k

$$\theta_i^{k+1} = \theta_i^k + \omega_i^{k+1} \quad (12.6)$$

where:

θ_i^{k+1}	Position of the i th particle at iteration $k + 1$
ω_i^{k+1}	Velocity of the i th particle at iteration $k + 1$

The searching process is continued until a relatively unchanged position has been encountered or computational limits are exceeded [2]. An important aspect of the PSO is that the ratios of the three elements that influence the particle velocity in the optimisation process can be modified. Therefore, the particle performance towards the optimal solution can be enhanced controlling the weighting coefficients [11].

12.5 Proposed Peer-to-Peer MATLAB–PowerFactory Communication

PowerFactory supports some external interfaces that can be used for data exchange and MATLAB coupling [32]. This chapter presents the P2P MATLAB–PowerFactory communication. The method is an extremely simple file sharing approach to couple MATLAB and PowerFactory, and it is used to solve an optimisation problem. The optimal tuning and placement of the PSS in a multi-machine power system using PSO algorithm is solved using the proposed P2P communication.

To perform the optimal tuning and placement of the PSS in a multi-machine power system, it is required to know the electromechanical eigenvalues of the power systems. Hence, the model analysis command (*ComMod*) included in DlgSILENT PowerFactory is needed, and it is an important computing task and provides a fast and precise tool for the eigenvalue calculation.

Generally, all evolutionary optimizers require a large number of function evaluation in order to reach the optimal or near-optimal solutions. In this chapter, the optimisation algorithm is based on a PSO algorithm which is implemented using a MATLAB script. On the other hand, the numerical values required to evaluate the objective functions and assess the constraints are obtained from the modal analysis

function in DIgSILENT PowerFactory. As a consequence, MATLAB and DIgSILENT should be linked together in a genuine automatic data exchange procedure.

The proposed P2P MATLAB–PowerFactory communication is customised to the PSO to solve the problem of optimal tuning and placement of the PSS in a multi-machine power system. However, the approach could be extended to any kind of calculation that requires the data interchange between MATLAB and DIgSILENT.

A P2P data exchange is as follows:

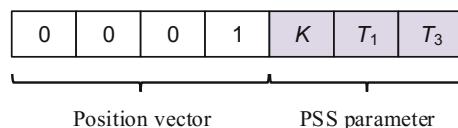
- A particle from the swarm is exported from MATLAB to DIgSILENT.
- DIgSILENT uses the data encrypted in the particle to compute the system eigenvalues.
- The corresponding system eigenvalues are exported from DIgSILENT to MATLAB.
- PSO uses the eigenvalues to evaluate the particle and to update the new particle position and speed.

The optimisation process involves the data interchange between MATLAB and DIgSILENT, making each particle passing through the above-described sequence. The P2P created a communication loop between the two software. The core of the communication between the software is the use of comma-separated values (.csv) files as a communication channel to make the information flow between the software. It is explained in greater detail in the next section.

The problem of the optimal tuning and placement of the PSS in a multi-machine power system is solved using a metaheuristic approach. Hence, each particle from the swarm is divided into two main parts: one dedicated to the location and the other to the setting. The first four elements of the particles (position vectors) are binary coded and contain information about the PSS position. The speed and position of the particles are computed and updated like in the case of the classical PSO; however, the data associated with the PSS position takes only the values 0–1. Considering, for example, that a random particle has the structure of the position vector according to Fig. 12.6 (the fourth case of the position vector has the value 1); it means that the PSS is installed into the excitation system of generator 4.

Because the speed of the particles corresponding to the position vectors increases their values according to the position vector of pbest and gbest, the position of the particle with the highest value is considered 1, and the others 0.

Fig. 12.6 Structure of a particle



12.6 Automatic Data Exchange Between MATLAB and DIgSILENT

The proposed P2P MATLAB–PowerFactory communication is designed to take the advantages of the both platforms in order to solve power system problems in a more handily manner. MATLAB is a software extremely convenient to perform complex mathematical operations and it is combined with the fast and accurate power system calculation method provided by DIgSILENT PowerFactory. The integration of both software opens the door for a powerful tool capable of coping with the challenges in the smart grids.

The proposed P2P MATLAB–PowerFactory communication procedure for the study case of this chapter is presented in Fig. 12.7. The automatic data exchange code is implemented using script in MATLAB and DIgSILENT Programming Language (DPL).

The data exchange is made using a link layer three buffer Comma-separated values (.csv) files:

- *Test1.csv* is used to transfer the particles from MATLAB to DIgSILENT.
- *Test2.csv* which is used for transferring the corresponding system eigenvalues from DIgSILENT to MATLAB.
- *Protocol.csv* is used for reading/write protocol of the involved.

For a fast and reliable communication, the protocol file (*Protocol.csv*) is intended to keep the control on the communication between the software; the data is based on a binary code taking the values 0 and 1.

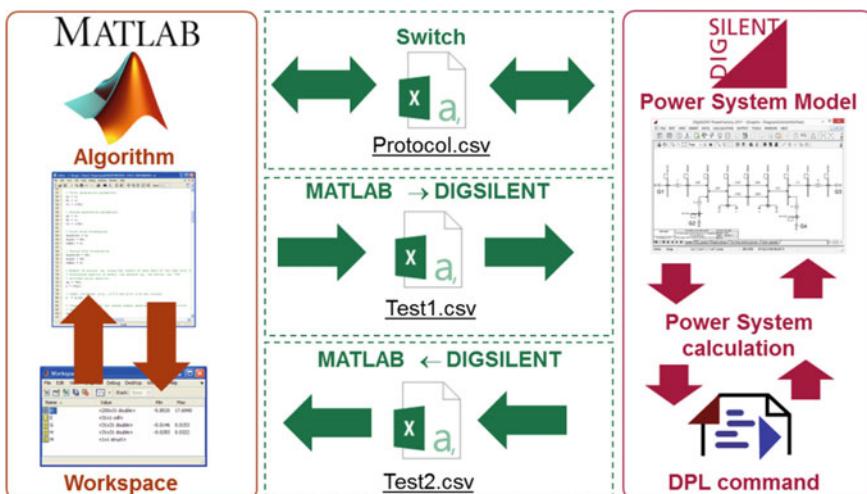


Fig. 12.7 Automatic data exchange between MATLAB and DIgSILENT

- If the value of *Protocol.csv* is 0, MATLAB reads the *Test2.csv* file, runs PSO, writes the *Test1.csv* file and DIgSILENT waits.
- If the value of *Protocol.csv* is 1, DIgSILENT reads the *Test1.csv* file, runs the modal analysis, writes the *Test2.csv* file and MATLAB waits.

During the P2P communication, both software scan and change the value of the *Protocol.csv* file after each of them finishes its work.

12.7 Simulation and Results

Initially, the small signal stability of the test system is evaluated by calculating the eigenvalues of the pure electromechanical system using modal analysis/eigenvalue calculation function in DIgSILENT.

The numerical results of the modal analysis, eigenvalues of the tests system considering the AVR but without PSS and governor is shown in Table 12.1. Three oscillation modes or three EO_s are identified. The oscillation mode 1 ($f_n = 1.0999$ Hz) is a local mode in which the generators from Area 1 are oscillating one against the other (G1 oscillate against G2). The mode of oscillation 2 ($f_n = 1.0644$ Hz) is a local oscillation mode in which the generators from Area 2 are oscillating one against the other (G3 oscillate against G4). The oscillation mode 3 ($f_n = 0.5361$ Hz) is an inter-area in which the generators from Area 1 are swinging against the generators from Area 2.

Figure 12.8 shows the location of the eigenvalues of the Tests System considering only the electromechanical components and the AVR. The two local oscillation modes exhibit a moderate damping (mode 2 and 3). The inter-area mode (mode 3) has a very low real component, providing a very low damping $\xi = 0.00781$, much smaller than is considered to be satisfactory in practice ($\xi > 0.05$). The relatively low real part of the inter-area eigenvalue could lead to large excursions during a small signal disturbance. As a consequence, the mode should be relocated using a PSS controller.

The proposed P2P approach is implemented and used to solve the problem of optimal tuning and placement of the PSS in the considered test system. The PSO algorithm is configured to a population of 150 particles and 80 iterations.

Table 12.1 Electromechanical eigenvalues of the test system considering the AVR and no other controller

Mode	Oscillation mode type	System eigenvalues	Frequency of oscillation (f_n , Hz)	Damping ratio (ξ)
1	Local mode	$-0.6100 \pm 6.9114j$	1.09999	0.087919
2	Local mode	$-0.5999 \pm 6.6883j$	1.06448	0.089338
3	Inter-area mode	$-0.02633 \pm 3.3690j$	0.53619	0.007817

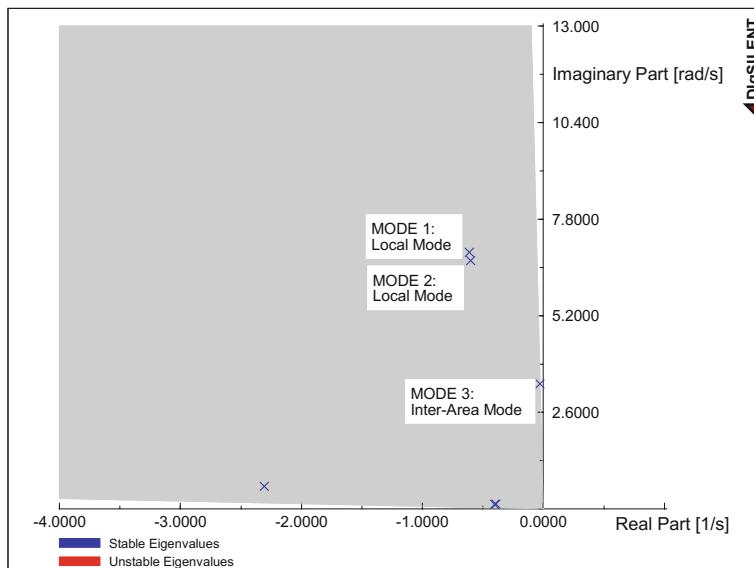


Fig. 12.8 Electromechanical eigenvalues of the test system considering the AVR and no other controller

Discussion about the parameter settings is beyond the scope of this chapter. However, a simple parameter setting procedure helped on the setting of the values of the weighting coefficients C_1 and C_2 and to the inertia coefficient W .

Consequently, these parameters must be optimally chosen in order to move the particle gradually throughout the searching space. After multiple runs, the results indicated that for this problem, the optimal values for C_1 and C_2 are 0.6 and 0.3, respectively. Then, W is decreased during the searching process in the interval (0.7–0.5), thus assigning to the optimisation process a greater local search ability near the end of the process.

The margin stability limits are defined by the values of σ_0 and ξ_0 , those values are settled for the test system as chosen -2.5 and 0.1 , respectively. Finally, a value of 10 is considered for the scaling factor (α) of the PSO algorithm.

The results of the optimal tuning and placement of the PSS in the test system indicate the optimal location of the PSS is at generator G3, and the specific setting of the controller parameters are shown in Fig. 12.8. The results of the modal analysis, eigenvalues of the tests system with the optimal location and setting of the PSS are shown in Table 12.2 (Fig. 12.9).

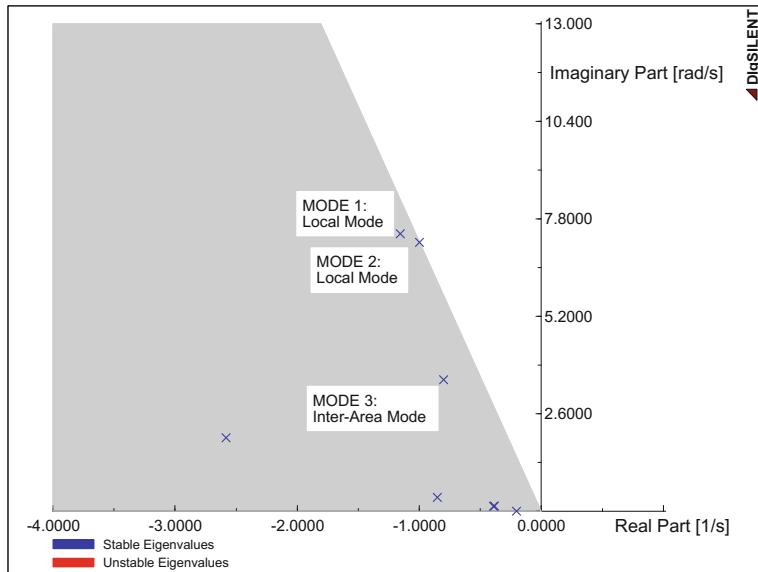
Numerical results in Table 12.2 show the improvement on the damping ratio of the inter-area mode (mode 3), but the frequency oscillation is not affected. Figure 12.10 shows the location of the test system eigenvalues conspiring the optimally designed controller to damp the inter-area oscillations, compared with Fig. 12.8, it is evident how the mode 3 is displaced to the left hand of the plane-s.

G3	K	T_1	T_3
0	0	1	0

Position vector PSS parameter

Fig. 12.9 Optimal particle**Table 12.2** Electromechanical eigenvalues of the test system considering PSS, AVR and governor

Mode	Oscillation-type	System eigenvalues with AVR + PSS	Frequency of oscillation (f_n , Hz)	Damping ratio (ξ)
1	Local mode	$-1.1553 \pm 7.4043j$	1.178434	0.154165
2	Local mode	$-0.9964 \pm 7.1723j$	1.141507	0.137609
3	Inter-area mode	$-0.7986 \pm 3.5087j$	0.558430	0.221944

**Fig. 12.10** Electromechanical eigenvalues of the test system considering the PSS and other controller

Comparing the results of the modal analysis without (Fig. 12.8) and with the optimal location and placement (Fig. 12.10) of the PSS in the test system, it is evident the eigenvalues of the test system with the optimal designed PSS are driven on the left side of the complex plane, and the damping ratio has improved considerably.

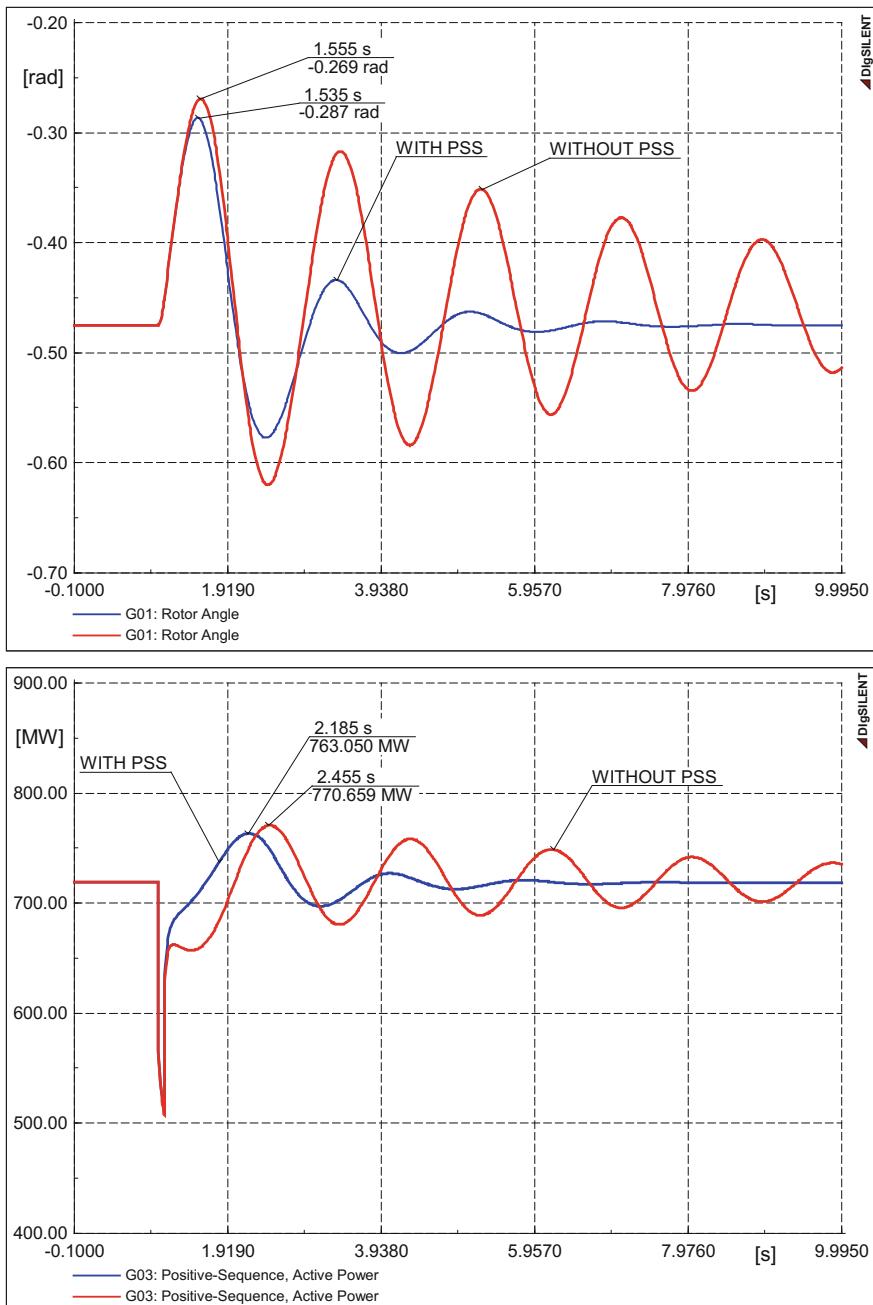


Fig. 12.11 Time-domain response of the test system: Three-phase short circuit in Line 7–8(1), cleared in five cycles

The improvement created by the optimally designed PSS is evident, but it should be noticed that the influence of the PSS upon the Local mode 1 of oscillation is minimal. It is because the participation factor of the generator G3 into the Local mode 1 of oscillation is very small.

On the other hand, the influence of G3 into Local mode 2 and Inter-area mode is significant. It is due to large participation factor of G3 into these oscillatory modes; it leads to large excursions of those eigenvalues in the left side of the complex s -plane.

A potential solution to the oscillatory mode on which the optimally designed PSS has a small influence can be the use of second PSS. This device is installed into the excitation system of the generator which has a large participation factor to that oscillatory mode.

To demonstrate the effectiveness and robustness of the proposed optimally designed PSS, the test system is subjected to further tests considering a fault. A line fault is used to excite the oscillatory modes in the test system; as a consequence, three-phase short circuits is inserted at $t = 1$ s at line 7–8(1), the three phase is successfully cleared five cycles later. The fault is cleared without opening the circuit breaker associated with the transmission line. The test system response is simulated using DIgSILENT PowerFactory, and time-domain simulations (RMS, ComSim) are used to evaluate the dynamic response of the system. The plots of the active power provided by generator G3 and the power angle at G1 are shown in Fig. 12.11.

The benefit of the optimally designed PSS can be easily observed by the obvious improvement on the damping related to the rotor angle and active power quantities shown in Fig. 12.11. Including the PSS allows damping the oscillation quick.

12.8 Conclusions

The increasing complexity of the power and energy domain due to the ongoing technical developments is creating many challenges in modelling and simulation of power systems: New domains, validation/de-risking difficult than before, and increased computational complexity. The co-simulation allows different subsystems, which form a coupled problem, to be modelled and simulated in a distributed manner. As a consequence, the co-simulation is becoming an integral part of validation; new domain interactions captured in more detail (realistic) and facilitate sharing of computational complexity. This chapter presents the Peer-to-Peer MATLAB–PowerFactory communication. The proposed method is based on a simple file sharing approach to couple MATLAB and PowerFactory. The proposed approach is demonstrated using a classical optimisation problem. An illustrative two-area power system is modelled used PowerFactory and an optimisation algorithm is implemented in MATLAB. The optimisation process is used for optimal tuning and placement of Power System Stabilizer (PSS) to enhance the power system stability. The optimisation algorithm used in this chapter is an evolutionary

algorithm (Particle Swarm Optimisation—PSO). MATLAB and DIgSILENT are employed and linked together in a genuine automatic data exchange procedure. The performance of the proposed PSO-based PSS test system in damping power system oscillations is proved through eigenvalue analysis and time-domain simulations.

References

1. G. Rogers, *Power System Oscillations*, Boston, [Mass.], vol. 11 (Kluwer Academic, London, 2000), p. 328
2. A. Stătivă, M. Gavrilaş, *Optimal PSS Setting for Improving Power System Stability*, presented at the FOREN 2012, Neptun, România, 2012
3. P. Kundur, N.J. Balu, M.G. Lauby, *Power System Stability and Control*, vol. 23 (New York, London, McGraw-Hill, 1994), p. 1176
4. B. Pal, B. Chaudhuri, *Robust Control in Power Systems* (Power electronics and power systems), vol. 25 (Springer, New York, 2005), p. 190
5. J. Machowski, J.W. Bialek, J.R. Bumby, *Power System Dynamics: Stability and Control*, 2nd ed. vol. 27 (Oxford, John Wiley, 2008), p. 629
6. J.L. Rueda, I. Erlich, F. Gonzalez-Longatt, Performance assessment of evolutionary algorithms in power system optimization problems, in *PowerTech*. IEEE Eindhoven 2015, 1–5 (2015)
7. A. Stătivă, M. Gavrilaş, V. Stăhie, Optimal tuning and placement of Power System Stabilizer using Particle Swarm Optimization algorithm, in *2012 International Conference and Exposition on Electrical and Power Engineering*, 2012, pp. 242–247
8. P. Korba, Real-time monitoring of electromechanical oscillations in power systems: first findings. *IET Gener. Transm. Distrib.* **1**(1), 80–88 (2007)
9. M.B. Saleh, M.A. Abido, Power system damping enhancement via coordinated design of PSS & TCSC in multimachine power system, in *2006 IEEE GCC Conference (GCC)*, (2006), pp. 1–6
10. J. Cepeda, J. Rueda, I. Erlich, A. Korai, F. Gonzalez-Longatt, Mean–Variance Mapping Optimization Algorithm for Power System Applications in DIgSILENT PowerFactory, in *PowerFactory Applications for Power System Analysis*, ed. by F.M. Gonzalez-Longatt, J. Luis Rueda, (Power Systems: Springer International Publishing, 2014), pp. 267–295
11. A. Stătivă., M. Gavrilaş, Using DIgSILENT in Small Signal Stability Studies, presented at the 5th International Symposium on Electrical Engineering and Energy Converters ELS 2013, Suceava, România, 26–27 Septembrie 2013, (2013)
12. J. L. Rueda-Torres, F. González-Longatt, *Data Mining and Probabilistic Power System Security*. (London: John Wiley & Sons Inc., 2017)
13. Y.N. Yu, Q.H. Li, Pole-placement power system stabilizers design of an unstable nine-machine system. *IEEE Trans. Power Syst.* **5**(2), 353–358 (1990)
14. P.S. Rao, I. Sen, Robust pole placement stabilizer design using linear matrix inequalities. *IEEE Trans. Power Syst.* **15**(1), 313–319 (2000)
15. G. Radman, Design of power system stabilizer based on LQG/LTR formulations, in *Conference Record of the 1992 IEEE Industry Applications Society Annual Meeting*, vol. 2 (1992), pp. 1787–1792
16. M.A. Abido, Y.L. Abdel-Magid, Robust design of multimachine power system stabilisers using Tabu Search Algorithm. *IEE Proc.—Gener. Transm. Distrib.* **147**(6), 387–394 (2000)
17. R. Asgharian, S.A. Tavakoli, A systematic approach to performance weights selection in design of robust H PSS using genetic algorithms. *IEEE Trans. Energy Convers.* **11**(1), 111–117 (1996)

18. M.A. Abido, Robust design of multimachine power system stabilizers using simulated annealing. *IEEE Trans. Energy Convers.* **15**(3), 297–304 (2000)
19. M.A. Abido, Optimal design of power-system stabilizers using particle swarm optimization. *IEEE Trans. Energy Convers.* **17**(3), 406–413 (2002)
20. J.L. Rueda, F. Gonzalez-Longatt, Application of Swarm Mean-Variance Mapping Optimization on location and tuning damping controllers, in *2015 IEEE Innovative Smart Grid Technologies—Asia (ISGT ASIA)*, (Bangkok, Thailand, 2015), pp. 1–5
21. A. Farah, T. Guesmi, H.H. Abdallah, A. Ouali, Optimal design of multimachine power system stabilizers using evolutionary algorithms, in *2012 First International Conference on Renewable Energies and Vehicular Technology*, (2012), pp. 497–501
22. A.A. Ba-muqabel, M.A. Abido, Review of conventional power system stabilizer design methods, in *2006 IEEE GCC Conference (GCC)*, (2006), pp. 1–7
23. F.L. Pagola, I.J. Perez-Arriaga, G.C. Verghese, On Sensitivities, Residues and Participations. Applications to Oscillatory Stability Analysis and Control. *IEEE Power Eng. Rev.* **9**(2), 61 (1989)
24. A. Karimpour, R. Asgharian, O.P. Malik, Determination of PSS location based on singular value decomposition. *Int. J. Electr. Power Energy Syst.* **27**(8), 535–541, 10/2005
25. H.F. Wang, F.J. Swift, M. Li, Indices for selecting the best location of PSSs or FACTS-based stabilisers in multimachine power systems: a comparative study. *IEE Proc.—Gener. Transm. Distrib.* **144**(2), 155–159 (1997)
26. J.L. Rueda, F. Gonzalez-Longatt, Location and Tuning of Damping Controllers by Using Swarm Mean-Variance Mapping Optimization. *Int. J. Eng. Sci. Technol. Special Issue: Energy, Econ. Environ.* **7**(3), 5 (2015)
27. C.E.A. Report, in *Ontario Hydro, Investigation of Low Frequency Inter-area Oscillation Problems in Large Interconnected Systems*, (1993)
28. M. Klein, G.J. Rogers, P. Kundur, A fundamental study of inter-area oscillations in power systems. *IEEE Trans. Power Syst.* **6**(3), 914–921 (1991)
29. IEEE guide for synchronous generator modeling practices and applications in power system stability analyses, *IEEE Std 1110-2002 (Revision of IEEE Std 1110-1991)*, (2003), pp. 1–72
30. IEEE recommended practice for excitation system models for power system stability studies, *IEEE Std 421.5-2005 (Revision of IEEE Std 421.5-1992)*, (2006), pp. 1–93
31. A.R. Akkawi, M.H. Ali, L.A. Lamont, L.E. Chaar, Comparative study between various controllers for power system stabilizer using Particle Swarm Optimization, in *2011 2nd International Conference on Electric Power and Energy Conversion Systems (EPECS)*, (2011), pp. 1–5
32. A. Latif, M. Shahzad, P. Palensky, W. Gawlik, An alternate PowerFactory Matlab coupling approach, in *2015 International Symposium on Smart Electric Distribution Systems and Technologies (EDST)*, (2015), pp. 486–491

Chapter 13

Implementation of the Single Machine Equivalent (SIME) Method for Transient Stability Assessment in DIGSILENT PowerFactory



Jaime Cepeda, Paúl Salazar, Diego Echeverría and Hugo Arcos

Abstract The development of novel smart transmission grid applications has recently gained deep interest based on the fact that there has been a wide deployment of technology capable of controlling the system in real time. In fact, some smart grid applications have been designed in order to perform timely Self-Healing and adaptive reconfiguration actions based on system-wide analysis, with the objective of reducing the risk of power system blackouts. In this new framework, real-time vulnerability assessment has to be firstly done in order to decide and coordinate the appropriate preventive or corrective control actions, such as special protection schemes. Since transient stability is, indeed, one of the most critical causes of system vulnerability, developing and applying assessment methodologies capable of delivering quick responses is fundamental among this smart structure. In this connection, a hybrid method, named Single Machine Equivalent (SIME), seems to

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_13) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

J. Cepeda (✉) · P. Salazar · D. Echeverría
Operador Nacional de Electricidad—CENACE, Av. Atacazo
and Panamericana Sur km 0, Quito, Ecuador
e-mail: cepedajaim@ieee.org

P. Salazar
e-mail: psalazar@cenace.org.ec

D. Echeverría
e-mail: decheverria@ieee.org

H. Arcos
Escuela Politécnica Nacional—EPN, Ladrón de Guevara E11-253,
Quito, Ecuador
e-mail: hugo.arcos@epn.edu.ec

present good perspectives of application for orienting both preventive control actions (off-line simulations) or corrective control actions (real-time analysis: emergency SIME). This method uses a combination of time-domain signals together with the equal area criterion (EAC) in order to determine the transient stability status based on the computation of stability margins. One of the challenges regarding the application of the SIME method is to adequately integrate it with a time-domain solver routine (such as the one already implemented in DIgSILENT PowerFactory). In this connection, this chapter addresses key aspects concerning the implementation of SIME by using DIgSILENT Programming Language (DPL). An application example on a well-known benchmark power system is then presented and discussed in order to highlight the feasibility and effectiveness of this implementation in DIgSILENT PowerFactory environment.

Keywords DIgSILENT Programming Language · DPL · SIME
Transient stability

13.1 Introduction

A number of factors such as transitioning energy policies, market pressures, slow transmission expansion, bulk power transfers over long distances and increasing environmental constraints have pushed power systems to be frequently operated close to their technical limits [1]. Under these conditions, some critical contingencies may lead to major consequences, including widespread disruptions or even blackouts, whose root causes are occasionally attributed to large rotor angle deviations [2]. As this regards, emerging intelligent technology has allowed designing the structure of the so-called smart grid, which is supposed to efficiently respond to the actual system conditions and provide autonomous control actions to enhance the system security in real time. This novel scheme requires special control strategies which should be adjusted depending on the real-time event progress. In this context, it is necessary to introduce a “Self-Healing Grid” functionality, which can provide critical information in real time, allow fast vulnerability assessment and perform timely and adaptive wide control actions. In general terms, vulnerability begins to develop in a specific region of the system (i.e. the vulnerable area), which is characterised by five different symptoms of system stress namely voltage instability, transient instability, poorly damped power oscillations, frequency deviations outside limits and overloads [3].

As mentioned, among the vulnerability symptoms, it is the transient stability (TS), which is a type of rotor angle stability that occurs when the system is subjected to a severe disturbance (e.g. short circuit on a transmission line). The time-frame of interest in this phenomenon is usually 3–5 s following the disturbance. This type of instability is usually characterised by a lack of synchronising torque [4]. Traditionally, transient stability assessment (TSA) has been conducted by using off-line dynamic simulations for a given set of credible contingencies. This framework has been widely used for design and tuning of protective and control systems and to provide guidelines for secure operations as well. By contrast,

real-time TSA is appropriate for evaluating the progress of actual transient phenomena occurring in a power system [5]. In this context, several smart grid applications have been recently developed to improve monitoring, protection and control tasks in real time. These kinds of applications are mainly based on emerging technologies such as phasor measurement units (PMUs), and wide area monitoring, protection and control systems (WAMPAC) [6–8]. The use of PMUs facilitates the measurement of electrical quantities at high sampling rates; so it allows tracking post-fault transient evolution in real time [6]. Moreover, post-contingency corrective control actions could also be executed within real time as long as reliable TSA is previously guaranteed [6].

Whether the selected assessment method is off-line or real time (depending on the subsequent control strategy has been conceived as preventive or corrective), a particular interest of the smart grid application is to consider the improvement of calculation and simulation time. This requirement is based on the fact that the opportunity of the control trigger is decisive for ensuring the system security. In this connection, a so-called hybrid method for TSA named Single Machine Equivalent (SIME) has the potential to give timely results suitable for smart grid applications. Basically, this method uses multi-machine rotor angles (δ_i) and the system accelerating power (P_a) for synthesising the so-called one machine infinite bus (OMIB) equivalent. The OMIB eases the prediction of transient instability from time-domain simulation results [9] (or equivalently from PMU data in case of real-time applications, in which case the method is named as emergency single machine equivalent E-SIME [5]).

Based on the previously stated facts, this chapter explores and discusses key aspects concerning the implementation of SIME method in DIgSILENT PowerFactory, based on the available resources of DIgSILENT Programming Language (DPL). The calculation stages involved in SIME, i.e. determination of OMIB equivalent, identification of critical and non-critical machines and the computation of the SIME relationships and stability margins, are structured in a master DPL script that subordinates several DPL subroutines. These DPL scripts can be employed, without major modifications, for analysing the TSA of any power system modelled in DIgSILENT PowerFactory. Besides, the chapter provides an application example of the SIME-based TSA on the IEEE New England 39-bus test system.

Following this introduction, the outline of the chapter is as follows: Sect. 13.2 presents a review of SIME theory. Section 13.3 overviews and discusses the implementation of the DPL scripts. Section 13.4 describes the application example and provides a discussion on the experimental results of the test case study. Finally, Sect. 13.5 summarises the main concluding remarks and outlines prospective future work.

13.2 The SIME Method

The Single Machine Equivalent (SIME) is a hybrid method resulting from the coupling of a time-domain transient stability program with the equal area criterion (EAC) [10]. It means that the time-domain method is applied to the multi-machine system under study, and the EAC is applied to a one machine equivalent that will

henceforth be called one machine infinite bus (OMIB) [11]. In this regard, the SIME method combines the step-by-step time-domain integration method applied to the multi-machine system and the EAC applied to the equivalent generator. This combination requires two basic steps: the identification of the critical machines (i.e. the generators responsible for the loss of synchronism) and the assessment of stability margins [12].

Below, a brief explanation of the methodology is presented.

13.2.1 Method Formulation

SIME relies on the following two propositions [13].

- *Proposition 1:* despite the complexity, the mechanism of a power system losing synchronism originates from the irrevocable separation of its machines into two clusters (critical machines and non-critical machines). Hence, the multi-machine system transient stability may be inferred from that of an OMIB system properly selected.
- *Proposition 2:* the transient stability of an OMIB may be assessed in terms of its transient stability margins, defined as the excess of its decelerating energy over its accelerating energy.

More specifically, SIME uses a generalised OMIB whose parameters are inferred from the multi-machine temporal data and refreshed at the same rate [15].

SIME concentrates on the post-fault configuration of a system subjected to a disturbance which presumably drives it to instability. The main stages of SIME are as follows: (i) separation of the machines into two groups; (ii) replacement of these clusters by two equivalent machines, and then by OMIB; and (iii) assessment the transient stability properties of this OMIB via the energy concept of EAC [15].

13.2.1.1 OMIB Formulation

With the usual notation, the i th machine parameters of an n machine system are denoted by [13]:

- δ_i, ω_i Rotor angle [rad], speed [rad/s]
- P_{mi}, P_{ei} Mechanical input power [pu], electric output power [pu]
- P_{ai} Accelerating power ($=P_{mi} - P_{ei}$) [pu]
- M_i Inertia coefficient [s].

To get an OMIB, first the machines are split into two clusters, say C (critical machines) and N (non-critical machines); afterwards, proceed as follows [13, 14]:

- (i) Transform the two clusters into two equivalent machines, using their corresponding partial centre of inertia (COI) angle, e.g. for cluster C , this results in:

$$\delta_C(t) = M_C^{-1} \sum_{k \in C} M_k \delta_k(t) \quad (13.1)$$

Similarly, for cluster N :

$$\delta_N(t) = M_N^{-1} \sum_{j \in N} M_j \delta_j(t) \quad (13.2)$$

where

$$M_C = \sum_{k \in C} M_k ; M_N = \sum_{j \in N} M_j \quad (13.3)$$

- (ii) Reduce this two-machine system into an equivalent OMIB system whose rotor angle is defined by:

$$\delta(t) = \delta_C(t) - \delta_N(t) \quad (13.4)$$

The corresponding OMIB rotor speed is defined in a similar way:

$$\omega(t) = \omega_C(t) - \omega_N(t) \quad (13.5)$$

where

$$\omega_C(t) = M_C^{-1} \sum_{k \in C} M_k \omega_k(t) ; \omega_N(t) = M_N^{-1} \sum_{j \in N} M_j \omega_j(t) \quad (13.6)$$

- (iii) Define the equivalent OMIB mechanical power by:

$$P_m(t) = M \left(M_C^{-1} \sum_{k \in C} P_{mk}(t) - M_N^{-1} \sum_{j \in N} P_{mj}(t) \right) \quad (13.7)$$

and the equivalent OMIB electric power by:

$$P_e(t) = M \left(M_C^{-1} \sum_{k \in C} P_{ek}(t) - M_N^{-1} \sum_{j \in N} P_{ej}(t) \right) \quad (13.8)$$

where M is the equivalent OMIB inertia coefficient, which is defined by:

$$M = \frac{M_C M_N}{M_C + M_N} \quad (13.9)$$

The resulting OMIB accelerating power is:

$$P_a(t) = P_m(t) - P_e(t) \quad (13.10)$$

Finally, with the above notations, the dynamics of the OMIB obeys to:

$$M \frac{d^2\delta}{dt^2} = P_m(t) - P_e(t) = P_a(t) \quad (13.11)$$

In this sense, the quantities P_m , P_e , P_a are obtained on the basis of the information provided by a time-domain simulation program (in this case from PowerFactory), to which SIME is coupled, which considers all the system and generator controls activated and uses Park equations. The OMIB's trajectory is not affected by any simplified hypothesis, except the one used by the time-domain simulation program. These quantities are computed for every time step of the simulation program [12].

13.2.1.2 Equal Area Criterion EAC

From a general point of view, the equal area criterion stipulates that the stability status of a dynamic system governed by an equation of the (13.11) type depends on the sign of margin η defined by [11]:

$$\eta = A_{dec} - A_{acc} \quad (13.12)$$

where A_{dec} (respectively A_{acc}) represents the decelerating area (respectively accelerating) (see Fig. 13.1): the system will be stable if η is positive, unstable if η is negative, the boundary between stability and instability taking place for $\eta = 0$.

The Eq. (13.12), which assesses the OMIB case using the equal area criterion, contains easy-to-compute particular expressions, as it will be solved below.

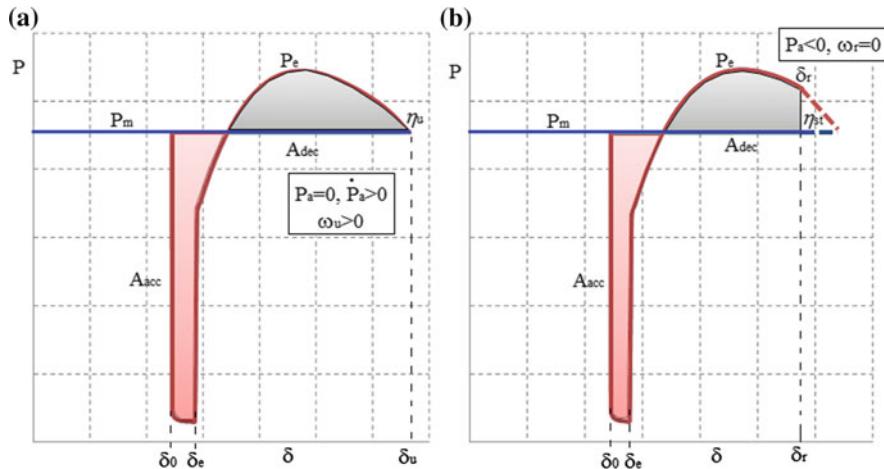


Fig. 13.1 OMIB Pa- δ representations: **a** unstable case, **b** stable case

13.2.1.3 Instability Criteria and the Corresponding Margin

An unstable OMIB trajectory reaches the “unstable angle” δ_u at the “unstable time” t_u when:

$$P_a(t_u) = 0 \quad \text{and} \quad \left. \frac{dP_a}{dt} \right|_{t=t_u} > 0 \quad (13.13)$$

This situation is satisfied with $\omega > 0$ for $t > t_u$, see Fig. 13.1.

The conditions (13.13) are “stopping conditions” for the time-domain simulation program provided by SIME. They mark the beginning of the loss of synchronism, and hence, any further computation is useless, except when a particular research is desired.

At the instant of time $t = t_u$, the stability margin η is defined by another very easy-to-compute expression:

$$\eta_u = -\frac{1}{2}M\omega_u^2 \quad (13.14)$$

13.2.1.4 Stability Criteria and the Corresponding Margin

A stable OMIB trajectory reaches its “recovery angle” $\delta_r < \delta_u$ at the “recovery time” t_r when the OMIB’s angle reaches its maximum value and then decreases [12], that is,

$$\omega(t_r) = 0 \quad \text{and} \quad P_a(t_r) < 0 \quad (13.15)$$

The conditions (13.15) are “the stopping conditions” of the SIME-based time-domain simulation program. They show whether the system is stable (at least as regards the first oscillation) and any further computation is useless unless the next oscillations are of interest.

At the instant $t = t_r$, the stability reserve η is given by (see Fig. 13.1):

$$\eta_{st} = \int_{\delta_r}^{\delta_u} |P_a| d\delta \quad (13.16)$$

It should be mentioned that unlike the instability margin defined by (13.14), the stability margin (13.16) can be determined only approximately. This vagueness happens because neither the angle δ_u nor the trajectory $P_a(\delta)$ with $\delta \in [\delta_r, \delta_u]$ is known since the OMIB trajectory “recovers” to stability condition, that is, the angle of the equivalent system starts decreasing after the maximum value $\delta = \delta_r < \delta_u$ is reached [12].

The following two approximations are proposed in [14]:

- a. The triangle approximation:

$$\eta_{st} \cong \frac{1}{2} P_a(\delta_u - \delta_r) \quad (13.17)$$

- b. The least squares approximation (weighted or not) [14], where the $P_a(\delta)$ curve is extrapolated on the interval $\delta \in [\delta_r, \delta_u]$. In this sense, this chapter proposes to use a quadratic regression, such as $P_a(\delta) = a\delta^2 + b\delta + c$, in order to extrapolate δ and determine δ_u .

13.2.1.5 Identification of the OMIB Equivalent

This identification is based on the following observation: “no matter how complex the system is, the transient instability phenomenon is triggered when the generators separate into two groups and the irreversible loss of synchronism results” [12]. This identification is based on *Proposition 1* in Sect. 13.2.1. The OMIB identification is carried out in the following manner [11]:

- (i) At every calculation step starting from t_e (time where the multi-machine system enters its final configuration), SIME ranks the machines in decreasing order of their rotor angles, then considers the first (for example, 10) “electric

- distances or intervals” between adjacent machines, ranked in decreasing order;
- (ii) Each of these “intervals” divides the machines into two groups, located on either side of it. SIME calculates the corresponding “OMIB candidate” and applies test (13.13) to it;
 - (iii) If one of the “OMIB candidates” fulfils the conditions in (13.13), it is considered the “real OMIB” (or OMIB in short); critical machines (and non-critical ones) are, respectively, those in groups above and below the interval. SIME calculates the corresponding margin (13.14) and stops the time-domain program;
 - (iv) If not, SIME decides the continuation of the time-domain program, proceeds to the calculation of the next step and repeats stages (i) and (ii) until it finds conditions (13.13) and executes step (iii).

Observation: The identification of the OMIB can be carried out only on an unstable trajectory. By continuation, the OMIB of the unstable trajectory identified earlier is considered still valid for a stable trajectory which is near enough to it (for example, obtained for a relatively close clearing time) [12].

13.2.1.6 Critical Machine Identification

The notion of critical machines is intimately related to unstable scenarios. By definition, on such an unstable multi-machine trajectory, the critical machines are those which go out of step, i.e. which cause the system loss of synchronism. To identify them, SIME drives, at each time step, the time-domain program first in the during-fault and then in the post-fault configuration. As soon as the system enters the post-fault phase, SIME starts recognising candidate decomposition patterns, until one of them reaches the instability conditions (13.13) defined by EAC [14].

More precisely, at each time step of the post-fault simulation, SIME sorts the machines according to their rotor angles, identifies the very first larger rotor angular deviations (“distances”) between adjacent machines and considers as candidate critical machines those who are above each one of these larger distances.

The methodology uses the data of rotor angles of synchronous machines, which are obtained from the time-domain simulations. In order to avoid a wrong selection of critical machines, the following formulation is applied, which allows normalising the rotor angles [3, 17]. A value $X_i(t)$ on bus i at time t is defined by (13.18), in this way:

$$X_i(t) = x_i(t) - x_{i0} - \bar{X} \quad (13.18)$$

where $x_i(t)$ represents the rotor angle on machine i at time t ; x_{i0} is its initial value before the perturbation, and (13.19) is the average value of the number of generators N_g at time t .

$$\bar{X} = \frac{\sum_{i=1}^{N_g} x_i(t)}{N_g} \quad (13.19)$$

An additional improvement of the proposed method for critical machine identification is made, as regards the classical method, where the influence of the generator's inertia in suggesting the critical and non-critical machines is considered. In order to identify whether the found set is critical or non-critical, an extra analysis is done. First, once the machines have been partitioned into two clusters, $C1$ and $C2$, their partial centres of inertia are determined as follows [17]:

$$M_{C1} = \sum_{i \in C1} M_i \quad (13.20)$$

$$M_{C2} = \sum_{j \in C2} M_j \quad (13.21)$$

where M_{C1} and M_{C2} are the partial centres of inertia; M_i and M_j are the inertia of each machine belonging to each cluster. In order to determine which cluster is critical or non-critical, the following inequality is written:

$$\begin{aligned} &\text{if } M_{C1} \geq M_{C2} \Rightarrow C1 \text{ is the non-critical machine cluster} \\ &\text{if } M_{C1} < M_{C2} \Rightarrow C1 \text{ is the critical machine cluster} \end{aligned} \quad (13.22)$$

The corresponding candidate OMIB parameters are computed according to expressions (13.1) to (13.10). The procedure is carried out until a candidate OMIB reaches the unstable conditions defined by (13.13); it is then declared to be the critical OMIB, or simply the OMIB (of concern).

13.2.2 Discussion About SIME Method

As concerning stability assessment, it is of interest to identify the instant when the instability is declared and its corresponding transient stability limit (i.e. critical clearing time or power limit), which usually requires considerably long time windows of domain simulations, and it is not possible to directly compute a stability margin. On the other hand, SIME declares instability just after passing the instability time t_u , i.e. after passing its unstable angle δ_u , and this is an unambiguous criterion since it, in fact, allows computing stability margins.

Time-domain programs rely on various theoretical criteria referring to a maximum angular deviation between extreme machines of the multi-machine system of, e.g. 180° or 360° ; or to a maximum deviation between one machine and a reference (e.g. the centre of angle or a heavy machine). Also, the time-domain approach is not

well adapted for fast filtering of contingencies since it does not provide stability margins that can tell “how far” the system is from instability and does not provide effective control tools, i.e. for stabilisation of unstable scenarios.

In summary, the main advantages of SIME method are listed below:

- Identification of the critical machines, which are useful for defining preventive and even emergency control actions;
- Assessment of the “time to instability”, which gives a clear indication of a system’s degree of instability;
- Additional representations of system dynamics, providing clear visualisation of complex transient stability phenomena; and
- Objective detection of instability/stability conditions that ensures criteria for early termination of time-domain simulations.

However, SIME presents some limitations in the case of very unstable simulation conditions, since OMIB might present a post-fault trajectory with only positive accelerating power, which precludes the computation of stability margins. This limitation may arise for the following two reasons: either because there is no intersection between the P_m and P_e curves or because the clearing angle is beyond the unstable angle. Also, another issue that needs taking care of is the implementation of adequate methods for critical machine identification. As mentioned in [17], this aspect is of main importance in transient stability since it allows making an adequate assessment, and mainly, it orients the execution of feasible control actions.

13.3 SIME Implementation in DPL Language

DIgSILENT Programming Language (DPL) offers an interface to develop automatic tasks in DIgSILENT PowerFactory. This functionality allows the creation of new user-defined calculation functions [16].

The main features of DPL comprise a particular programming language, similar to C++ that offers arithmetic and standard functions availability, as well as other types of control statements or logical operators (e.g. logic functions, loops, conditionals), including the handling of vectors and matrices.

The DPL command object (*ComDpl*) constitutes the central element. This object allows the connection of different parameters, variables or objects to various functions or internal elements in order to give results or change parameters.

The input to the DPL script can be predefined input parameters, single objects from the single line diagram, elements of the database or a set of objects or elements [16]. This input information can then be evaluated using functions and internal variables inside the script. Also, other PowerFactory objects can be defined and executed inside the DPL object as internal objects (i.e. calculation commands, subscripts also released in DPL, filter sets).

The DPL script is designed to run a set of operations to communicate itself with the database in order to read and/or change settings, parameters or results directly in the database objects. This functionality is used to implement the SIME algorithm in order to carry out a number of predefined time-step simulations (simulation by intervals) inside a loop. After each simulation, the system time-domain results are collected in order to trace the EAC of the system and determine the stability margins. This analysis is repeated until the system is determined to be stable or unstable. For accomplishing this application, a set of the system generators to be analysed is firstly specified via the definition of one *General Set* that contains both the synchronous machine elements (*ElmSym*) and the synchronous machine types (*TypSym*), using the subroutine *SIME_initialization.ComDpl*. Each element of this set is accessed by the DPL script in order to obtain its main variables at each

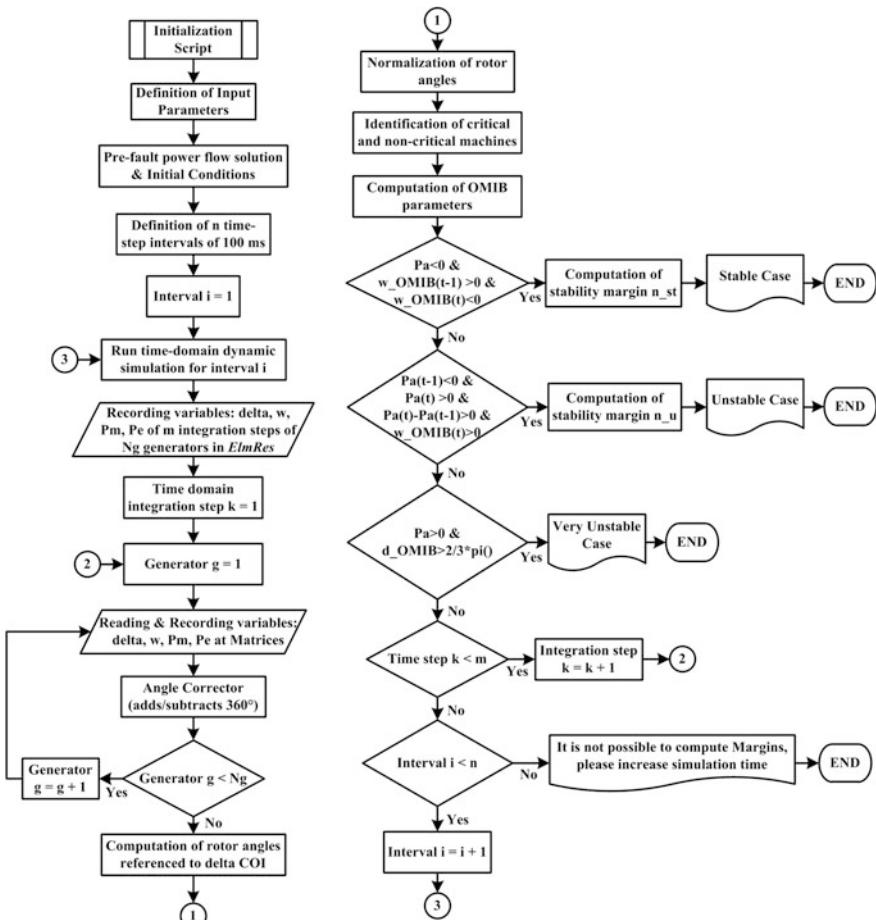


Fig. 13.2 SIME method algorithm

simulation (i.e. inertia coefficients, nominal power, rotor angles, electric power, speed). Inside the loop of the simulation by intervals, the main simulation engine is the time-domain solver represented by the *run simulation* command (*ComSim*).

The SIME method is implemented following the steps presented in Fig. 13.2. The following subsections describe, in detail, the implementation of the SIME method into DPL.

13.3.1 Initialisation Script

Before running the SIME algorithm, it is necessary to define the n generators (and their main variables) whose transient stability status will be analysed via SIME. For this purpose, a subroutine for initialising the application needs to be firstly run. This initialisation subroutine is *SIME_initialization.ComDpl* that communicates with one *General Set* whose elements are the *ElmSym* objects and the *TypSym* objects, which need to be previously defined. For this purpose, and in order to allow a sequential management of the entire algorithm programmed in the DPL scripts, both the *ElmSym* objects and their corresponding *TypSym* objects need to be named keeping a specific format code as follows: $G_p_\ast.Elmsym$, and $G_p_\ast.Typsym$, being p the corresponding number of the generator G ($p = 1, 2, 3, \dots, n$). Afterwards, the *ElmSym* objects and their associated *TypSym* objects that belong to the target generators (which can be a subset of the total generators) must be added to the *General Set Set_Gen_Sym_Typ.SetSelect* that is part of the *Contents* of the initialisation script. Finally, the subroutine *SIME_initialization.ComDpl* must be executed.

13.3.2 Main Script—SIME Implementation Procedure

The main script that controls the SIME method in PowerFactory is the DPL command *SIME_10G_39B.ComDpl*. In this script, the main dialogue (*Input parameters*) is used in order to specify the basic system parameters for simulating a TS study based on SIME method. These parameters can be modified by the user.

Figure 13.3 presents the *basic options* dialogue of *SIME_10G_39B.ComDpl*, where the *Input parameters* section contains the following features:

- (i) The clearing time of the simulated fault (*start*),
- (ii) The final simulation time (*end*),
- (iii) The system power base that is the reference for internal per unit analysis (*s_base*) and
- (iv) The number of generators whose rotor angles will be plotted after the complete algorithm finishes running (*num_gen_plot*). In order to ensure plots slightly, this last variable *num_gen_plot* can only take values between 2 and 10.

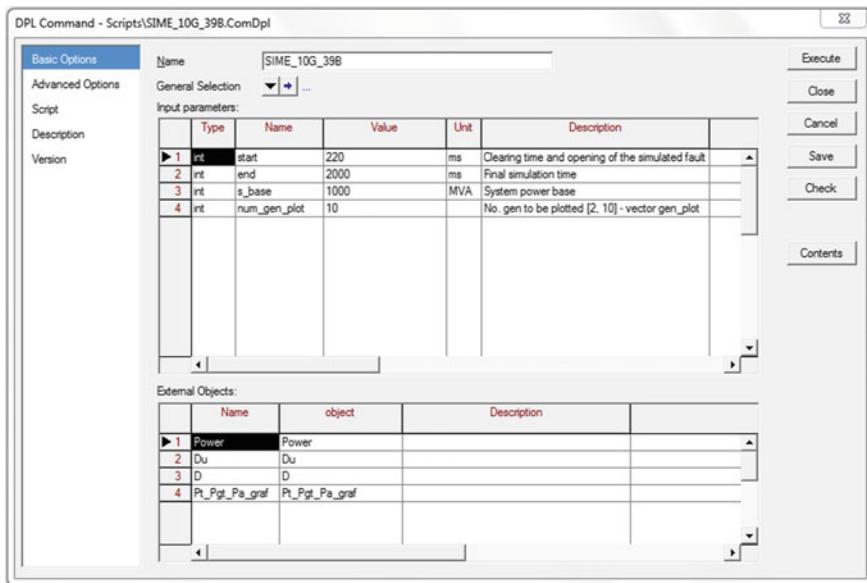


Fig. 13.3 SIME DPL command basic options

The main programming variables are included in the DPL programming code (*Script* section), whereas several internal objects, such as vectors (*IntVec*), matrices (*IntMat*), results (*ElmRes*) and the different subroutines (*.*ComDpl*), are included in the *Contents* section. Likewise, in the *External Objects* section, several vectors and matrices, that are part of the subroutines, are addressed in order to have them available in the main script.

There are two objects included in the *Contents* section that necessarily needs to be updated by users, before execution of the main script. The first one is the *Events*.*IntEvt* object that contains the specification of the fault to be simulated, i.e. the branch to be faulted, the fault event type as well as the fault inception time, the fault clearing time and the branch opening time (and all the interesting events to be analysed). One aspect to be highlighted is that the value of the fault clearing time specified in the *Events*.*IntEvt* object needs to be also filled in the variable *start* belonging to the *Input parameters* section. This requirement ensures that the SIME algorithm starts its computation at the moment of the clearing time, which is a basic conceptual condition in order to avoid wrong analysis during the fault period. The second one is the vector *gen_plot*.*IntVec* which has a length of 10 elements and

	Name	Order	Type	Object modified	Qt
gen_plot				7/30/2016 11:35:49 AM	
Events				7/30/2016 11:35:13 AM	
M_gen				7/30/2016 11:34:22 AM	
S				7/30/2016 11:34:18 AM	
FP				7/30/2016 11:34:14 AM	
R_simul_2				7/30/2016 11:31:57 AM	
simul	-1000000			7/30/2016 9:36:11 AM	
cal_fig	-1000000			7/30/2016 9:27:44 AM	
R_export	-1000000			7/30/2016 9:21:33 AM	
cic	-1000000			7/30/2016 9:21:33 AM	
export	-1000000			7/30/2016 9:21:33 AM	
ldf	-1000000			7/30/2016 9:21:33 AM	
reg_cua	-1000000			7/30/2016 9:21:33 AM	
sort_a2	-1000000			7/30/2016 9:21:33 AM	
sort_dif2	-1000000			7/30/2016 9:21:33 AM	
M				7/30/2016 9:21:33 AM	
PARAMETERS				7/30/2016 9:21:33 AM	
P_BASE				7/30/2016 9:21:33 AM	

Fig. 13.4 Input parameter: *gen_plot*, *Events*, *S*, *M_gen*, *FP*

contains the numbers p of the generators G whose rotor angles are interested to be plotted (this vector is strongly related with the variable *num_gen_plot*). These two objects are highlighted in Fig. 13.4.

In addition to the already mentioned objects, other basic input data for the SIME algorithm are those comprised of the elements of the following vectors: *S.IntVec* (Nominal Apparent Power [MVA]), *M_gen.IntVec* (inertia coefficient—rated to Nominal Apparent Power [s]) and *FP.IntVec* (Power Factor), which are also shown in Fig. 13.4. Remember that inertia coefficient “M” is twice constant inertia “H”. In this specific case, although these vectors might be filled in manually, the execution of the initialisation script *SIME_initialization.ComDpl* automatically updates these three vectors, so the user should not worry about changing them manually.

The DPL script has been structured in order to incorporate all the stages presented in the *SIME_10G_39B.ComDpl* search procedure depicted in Fig. 13.2.

First, all the internal variables have to be declared, as follows:

```
!!! SIME_10G_39B internal variable declaration
int cases,i1,i2,i3,i4,ele,indi,c1,c2;
int accountant,checker,condition;
int nn,nn_max,nn_min,simulation_time;
string parameters,generador,nom,name;
double Obs_phi,Obs_pt,Obs_pgt,Obs_speed,time,phi;
double
MT,Mg,Mg_n,M_gen_pu_n,deltacoi,omegacoi,delta_n,omega_n,delta_pu,dif,sum_Mxdelta,sum_M
xomega;
double w,pt,pgt,pa;
double s_g,fp_g,p_base,pgt_real,pt_real;
double delta_ini,sum_delta_rad,delta_ave;
double coi_n,coi_w_n,deltacoi_n,omegacoi_n,phi_coi;
double d_C,d_N,d_OMIB,w_C,w_N,w_OMIB,t_simul,M_OMIB;
double wu,nu,nst,du;
double t_win,osc,start_time,end_time,process_time;
double a,b,c;
double w_OMIB_ante,pa_ante,pa_resta;
double current_deltarad,previous_deltarad,difference_deltarad,correction_deltarad;
double deltarad_n;
```

To accelerate simulations, the user interface can be frozen, as follows:

```
Time_o=GetTime(4);    !Defining the initial processor time
EchoOff();           !Freezing the user-interface

...      !DPL script routines

simulation_time=(inicio/10)+10;
end_time=GetTime(4); !Defining the final processor time
process_time=end_time-start_time;
printf('%s %g','Processing time',process_time);   !Printing the elapsed time
EchoOn();  !Re-activating the user interface
```

After all, calculations restart, power flow is solved, and the initial conditions are calculated.

```

...      !DPL script routines

ResetCalculation();           ! Reset all calculations
ldf.Execute();                ! Calculate power flow
cic.Execute();                ! Calculate initial conditions

cal_fig:num_gen_plot = num_gen_plot; ! Selection of generators to be plotted

! Conversion pu System
nc=M_gen.Size();
M.Resize(nc);
M.Init(nc);
MT=0;
Mg_n=0;
for (indi=1;indi<=(nc);indi+=1){
  Mg=M_gen.Get(indi);
  s_g=S.Get(indi);
  M_gen_pu_n=Mg*s_g/s_base;
  M.Set(indi,M_gen_pu_n);
  fp_g=FP.Get(indi);
  p_base=s_g*fp_g;
  P_BASE.Set(indi,p_base);
  MT=MT+M_gen_pu_n;
}
...

```

Simulation intervals are defined in 100 ms (milliseconds), and time-domain dynamic simulation starts, recording the results in the *R_simul_2.ElmRes* object.

```

...      !DPL script routines

! Define of time windows for dynamic simulations
osc=start/1000;
nn_min=start/10;
nn_max=end/10;
simulation_time=start/10;
tstop=nn_min/10;
simul:tstop=tstop;

! Start dynamic simulations
for (nn=nn_min;nn<=(nn_max);nn+=10){
  printf('%s %g','simulation time requested (ms)=',nn*10);
  t_win=(nn)/100;
  tstop=t_win;
  simul:tstop=tstop;
  simul.Execute();          ! Run Simulation
  LoadResData(R_simul_2);   ! Load Data of simulation
}
...

```

In order to select the main electric values, the generators are searched (from the set defined by the initialisation), filtered and chosen, according to the name.

```
...      !DPL script routines

! Search and filter generators, according to the name
!! Start loop "ele"
for(ele=0;ele<nf-1;ele+=1){
  !! Start loop "cases"
  for(cases=1;cases<=(nc/4);cases+=1){
    generador=sprintf(' %s%d%s%', 'G_',(cases), '_', '*.ElmSym');
    Genx=SetObj.FirstFilt(generador);
    ! Choosing variables of interest
    if (Genx){
      nombre=sprintf('%s',Genx);
      nom = Genx.GetFullName();
      i1 = ResIndex(R_simul_2, Genx, 's:phi');
      i2 = ResIndex(R_simul_2, Genx, 's:pt');
      i3 = ResIndex(R_simul_2, Genx, 's:pgt');
      i4 = ResIndex(R_simul_2, Genx, 's:speed');
      GetResData(Obs_phi,R_simul_2,ele,i1);
      GetResData(Obs_pt,R_simul_2,ele,i2);
      GetResData(Obs_pgt,R_simul_2,ele,i3);
      GetResData(Obs_speed,R_simul_2,ele,i4);
      GetResData(tiempo,R_simul_2,ele,-1);
      deltarad_ori.Set(ele+1,cases,Obs_phi);
      deltarad.Set(ele+1,cases,Obs_phi);
    }
  }
}

...      !DPL script routines
```

In the time-domain simulations, the rotor angles of the generators are always referenced to a particular machine, usually selected as the slack. When an angle difference exceeds 180° (π radians), the complementary angle is computed and displayed, i.e. adds/subtracts $360^\circ \pm (2\pi$ radians) to the corresponding rotor angle, and then, the rotor angles need to be firstly corrected in order to avoid this reference change.

```
...      !DPL script routines

! ** Corrector upper / lower angles | 2 * pi | [ Rad ] )
  ! ** corrected matrix [deltarad] ** !
  if (ele>1){
    actual_deltarad=deltarad.Get(ele+1,cases);
    anterior_deltarad=deltarad.Get((ele),cases);
    diferencia_deltarad=(actual_deltarad-anterior_deltarad);
    if (abs(diferencia_deltarad)>6.1){
      if (diferencia_deltarad<0){
        correccion_deltarad=(actual_deltarad+(2*pi()));
        deltarad.Set(ele+1,cases,correccion_deltarad);
      }
      else {
        correccion_deltarad=(actual_deltarad-(2*pi()));
        deltarad.Set(ele+1,cases,correccion_deltarad);
      }
    }
  }

..      !DPL script routines
```

The real rotor angles, referenced to delta COI (centre of inertia), are computed.

```
...      !DPL script routines

! ** Referencing delta angles with respect to delcoi
! ** delta-delta_coi
!! Start loop "c1"
for (c1=1;c1<=(nc/4);c1+=1){
    delta_n=deltarad.Get(ele+1,c1);
    omega_n=W.Get(ele+1,c1);
    coi_n=coi.Get(ele+1);
    coi_w_n=coi_w.Get(ele+1);
    deltacoil_n=delta_n-coi_n;
    omegacoil_n=omega_n-coi_w_n;
    deltarad_coi.Set(ele+1,c1,deltacoil_n);
    W_coi.Set(ele+1,c1,omegacoil_n);
    delta_grad_coi.Set(ele+1,c1,deltacoil_n*180/pi()));

! Getting delta_rad initial value (first row)
delta_ini=deltarad_coi.Get(1,c1);
delta_inicial.Set(c1,delta_ini);
! Calculating the average of the angles for each row
sum_delta_rad=sum_delta_rad+deltacoil_n;
}
!! End loop "c1"
delta_pro=sum_delta_rad/(nc/4);
delta_promedio.Set(ele+1,delta_pro);

..      !DPL script routines
```

As mentioned in Sect. 13.2, to avoid a wrong selection of all critical machines, Eqs. (13.18) and (13.19) can be applied in order to firstly normalise the angles.

```

...
!DPL script routines
    ! ** Normalizing all angles in the matrix "delta"
    delta_n=0;
    delta_ini=0;
    delta_pro=0;
    delta_pu=0;
    coi_n=coi.Get(ele+1);
    for (c2=1;c2<=(nc/4);c2+=1){
        delta_ini=delta_inicial.Get(c2);
        delta_pro=delta_promedio.Get(ele+1);
        delta_n=deltarad_coi.Get(ele+1,c2);
        delta_pu=delta_n-delta_ini-delta_pro;
        delta.Set(ele+1,c2,delta_pu);
    }
    sum_delta_rad=0;
    sum_Mxdelta=0;
    sum_Mxomega=0;

    dif=0;
    deltarad_f.Resize(nc/4);
    deltarad_coi_f.Resize(nc/4);
    dif_phi.Resize((nc/4)-1);
    w_f.Resize(nc/4);
    pt_f.Resize(nc/4);
    pgt_f.Resize(nc/4);

    contador=0;
    verificador=0;
    condicion=-1;
    t_simul=(200/1000);
    paso_simul=t_simul/0.01;
    contador=contador+paso_simul+1;

    for(indi=0;indi<=((nc/4)-1);indi+=1){
        phi=delta.Get(ele+1,indi+1);
        phi_coi=deltarad_coi.Get(ele+1,indi+1);
        w=w_coi.Get(ele+1,indi+1);
        pt=pt.Get(ele+1,indi+1);
        pgt=Pgt.Get(ele+1,indi+1);
        deltarad_f.Set(indi+1,phi);           ! normalized angle
        deltarad_coi_f.Set(indi+1,phi_coi);
        w_f.Set(indi+1,w);
        pt_f.Set(indi+1,pt);
        pgt_f.Set(indi+1,pgt);
    }
...
!DPL script routines

```

Critical machines (CMs) are closely related to unstable scenarios; therefore, they are those that cause loss of synchronism or “out of step”. To identify them, the SIME methodology uses time-domain simulations corresponding to the power system under the post-fault condition. In this scenario, the SIME methodology begins the analysis at each iteration time of the solution of the differential-algebraic equations (DAE).

In more detail, at each iteration of the post-fault simulation period, the methodology instructs SIME generators according to the behaviour of the rotor angles; identifying the larger angular deviations between adjacent machines; and comparing these differences in order to determine the critical machines CM.

```
...      !DPL script routines

! ** Identification of critical machines
sort_a2.Execute();    ! Ordering angles
sort_dif2.Execute(); ! Ordering of angular differences between adjacent
generators
..      !DPL script routines
```

The stable condition is detected when: the accelerating power is negative and the sign of angular speed (omega) changes.

```
...      !DPL script routines

!! Start If condition stable
if ({pa<0}.and.{w_OMIB_ante>0}.and.{w_OMIB<0}){
  condicion=1;
  t_st=Tiempo.Get(ele+1);
  d_st=Pt_Pgt_Pa.Get(ele+1,6);
  if ({condicion=1}.and.{t_st>osc}){
    PARAMETERS.Set(1,nf);
    PARAMETERS.Set(2,nc);
    PARAMETERS.Set(3,paso_simul);
    PARAMETERS.Set(4,t_critico);
    PARAMETERS.Set(5,d_critico);
    PARAMETERS.Set(6,ele+1);
    PARAMETERS.Set(7, t_st);
    PARAMETERS.Set(8, d_st);
    PARAMETERS.Set(9, ele+1);
    end_time=GetTime(4);
    cal_fig.Execute();
    d_st=Pt_Pgt_Pa_graf.Get(ele+1,4);
    PARAMETERS.Set(8, d_st);
    reg_cua.Execute();      !! Start quadratic regression
    du=Du.Get(1);
    printf('s','STABLE CASE');
    printf('s %g','Pa(pu)=',pa);
    printf('s %g','w_OMIB(pu)=',w_OMIB);
    printf('s %g','recovery time tr (ms)=',t_st*1000);
    printf('s %g','recovery angle delta_r (rad)=',d_st);
    printf('s %g','recovery angle delta_r (degrees)=',d_st*(180/pi()));
    printf('s %g','unstable angle delta_u (rad)=',du);
    printf('s %g','unstable angle delta_u (degrees)=',du*(180/pi()));
    a=D.Get(3,1);
    b=D.Get(2,1);
    c=D.Get(1,1);
    nst=(a*pow((d_st-du),3)/3)+(b*pow((d_st-du),2)/2)+c*(d_st-du);
    printf('s %g','stability margin n_st=',nst);
    process_time=end_time-start_time;
    printf('s %g','Processing time',process_time);
    exit();
  }
  !! End If condition "stable"
..      !DPL script routines
```

The unstable condition is detected when: the sign of accelerating power changes, the difference $P_a(t) - P_a(t - 1)$ is positive and the speed (omega) is positive. In addition, an additional condition to identify very unstable cases is also included.

```

...      !DPL script routines

    !! Start If condition unstable
    if ({pa_antec<0}.and.{pa>0}.and.{pa_resta>0}){
        condicion=0;
        if ({w_OMIB>0}){
            t_critico=Tiempo.Get(ele+1);
            d_critico=Pt_Pgt_Pa.Get(ele+1,6);
            wu=w_OMIB;
            nu=-(1/2)*wu*wu*(2*pi()*60)*(2*pi()*60);
            if ({condicion=0}.and.{t_critico>osc}){
                PARAMETERS.Set(1,nf);
                PARAMETERS.Set(2,nc);
                PARAMETERS.Set(3,paso_simul);
                PARAMETERS.Set(4,t_critico);
                PARAMETERS.Set(5,d_critico);
                PARAMETERS.Set(6,ele+1);
                PARAMETERS.Set(7, t_st);
                PARAMETERS.Set(8, d_st);
                PARAMETERS.Set(9, ele+1);
                end_time=GetTime(4);
                cal_fig.Execute();           ! Start making graphic
                printf('%s','UNSTABLE CASE');
                printf('%s %g','Pa(pu)=',pa);
                printf('%s %g','w_OMIB(pu)=',w_OMIB);
                printf('%s %g','unstable time tu (ms)=',t_critico*1000);
                printf('%s %g','unstable angle delta_u (rad)=',d_critico);
                printf('%s %g','unstable angle delta_u
                                (degrees)=',d_critico*(180/pi()));
                printf('%s %g','stability margin n_u=',nu);
                process_time=end_time-start_time;
                printf('%s %g','Processing time',process_time);
                exit();
            }   !*** fin if condicion 0
        }
    }
    !! End if condition "unstable"

    !! Start If condition "very unstable"
    if ({pa>0}.and.{d_OMIB>2/3*pi()}){
        t_critico=Tiempo.Get(ele+1);
        d_critico=Pt_Pgt_Pa.Get(ele+1,6);
        wu=w_OMIB;
        PARAMETERS.Set(1,nf);
        PARAMETERS.Set(2,nc);
        PARAMETERS.Set(3,paso_simul);
        PARAMETERS.Set(4,t_critico);
        PARAMETERS.Set(5,d_critico);
        PARAMETERS.Set(6,ele+1);
        PARAMETERS.Set(7, t_st);
        PARAMETERS.Set(8, d_st);
        PARAMETERS.Set(9, ele+1);
        end_time=GetTime(4);
        cal_fig.Execute();           ! Start making graphic
        printf('%s','VERY UNSTABLE CASE');
        printf('%s %g','Pa(pu)=',pa);
        printf('%s %g','w_OMIB(pu)=',w_OMIB);
        process_time=end_time-start_time;
        printf('%s %g','Processing time',process_time);
        exit();
    }

..      !DPL script routines

```

There is the possibility of not computing any stability margins due to a lack of enough information. In these cases, it is necessary to increase the final simulation time (*end*) in the *Input parameters* section (Fig. 13.3). If this is the case, an informative message will be printed into the output window.

```
...     !DPL script routines
if (nn_max-nn<10){
    printf('%s','IT IS NOT POSSIBLE TO COMPUTE MARGINS, PLEASE INCREASE
SIMULATION TIME');
    exit();
}
...
!DPL script routines
```

Finally, as shown in the previous scripts, when the complete algorithm implemented in the DPL scripts finishes running, the main results are reported in the PowerFactory *Output Window*. These results include the following features:

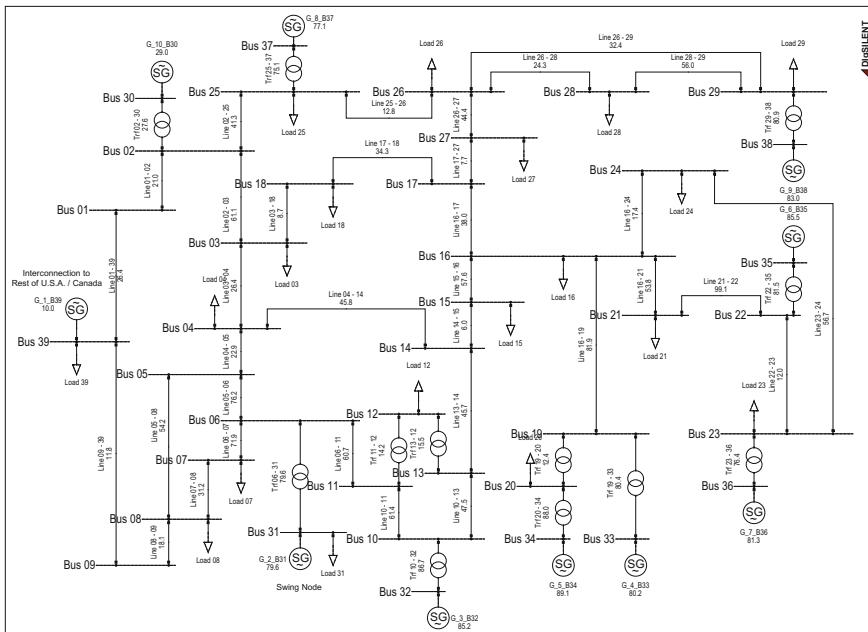


Fig. 13.5 IEEE New England 39-bus test system single-line diagram

- (i) Each one of the predefined time-step simulations (simulation by intervals) that should run inside the loop until the SIME stopping conditions (stability or instability criteria) are reached,
- (ii) The critical machines,
- (iii) The final stability status (*stable or unstable case*),
- (iv) The final OMIB variables (accelerating power P_a and angular speed ω),
- (v) The recovery time t_r and the extrapolation of the unstable time t_u (for stable cases) or the unstable time t_u (for unstable cases),
- (vi) The recovery angle δ_r and the extrapolation of the unstable angle δ_u (for stable cases) or the unstable angle δ_u (for unstable cases), in radians and degrees,
- (vii) the stability margin (n_{st} for stable cases or n_u for unstable cases) and
- (viii) the elapsed *processing time*.

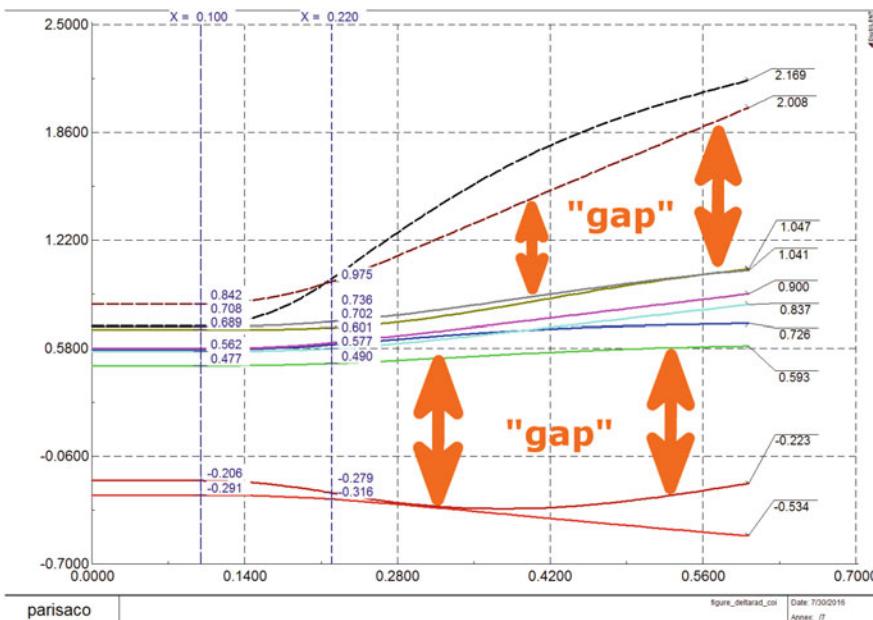


Fig. 13.6 Critical machines variation over time evolution

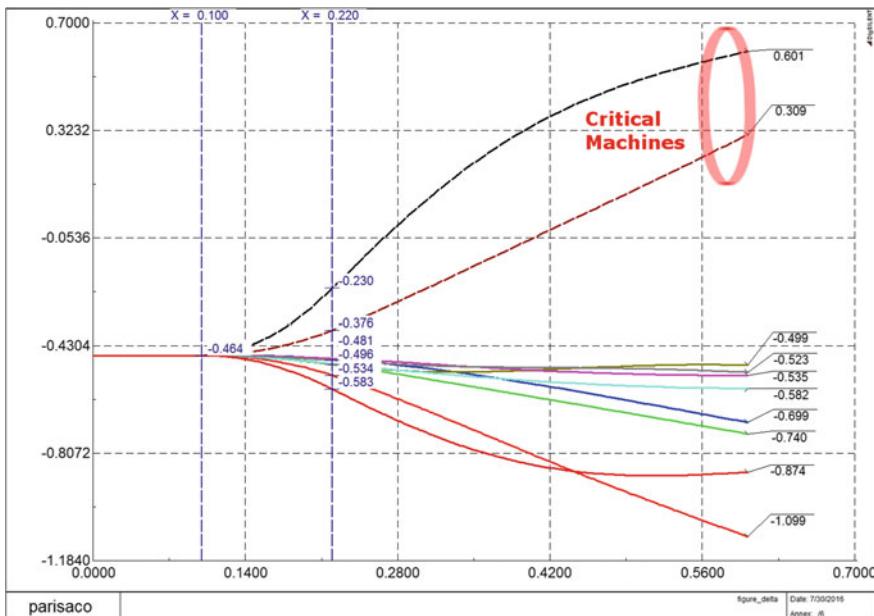


Fig. 13.7 Time evolution of normalised rotor angles

13.4 Application Example

In this section, an illustrative example regarding transient stability assessment is applied to a well-known benchmark power system. The results obtained from SIME are presented in order to highlight the potential of this tool to analyse the system TS status quickly.

13.4.1 Test System

The IEEE 39-bus 10-machine test system has been used in the simulations. All generators (except the one representing the interconnection with the New England System) are equipped with an automatic voltage regulator (AVR) *IEEE Type 1 Excitation System IEEET1*. The test system single-line diagram is depicted in Fig. 13.5.

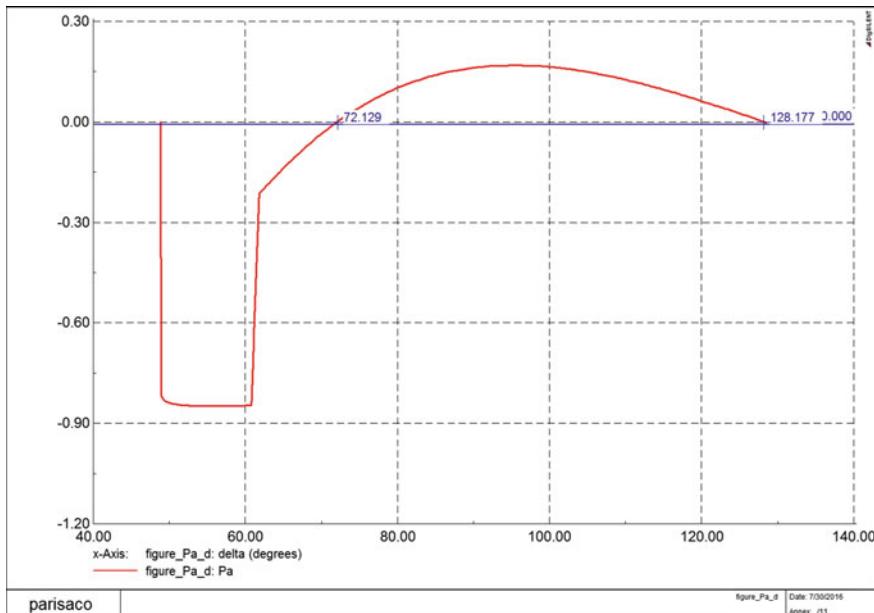


Fig. 13.8 OMIB accelerating power [pu] versus OMIB rotor angle [degrees]

13.4.2 Simulation Results—Unstable Case

A three-phase short circuit is applied on line 2–25 at $t = 0.1$ s, followed by the opening of the corresponding transmission line at $t = 0.22$ s (i.e. fault clearing time). The results obtained from the application of the developed SIME DPL scripts are presented as follows.

13.4.2.1 Critical Machines Identification

The conventional methodology for determining critical machines is to track the post-fault trajectory of generator rotor angles depending on time evolution. In this connection, at every integration time step, the rotor angles are sorted from highest to lowest and the greater angular deviation (larger gap) between two adjacent machines is determined. It is considered that the candidates for determining the

critical machines are those generators whose rotor angles are above the largest difference [17].

The methodology described in the preceding paragraph is suitable for cases where critical machines do not change over time, i.e. when the largest difference or gap is kept over time. Figure 13.6 shows an example of the variation of critical machines over time evolution, where it is worth to highlight that while time evolves, the presented gaps can eventually suggest a change in the coherency of generators. Thus, in this case, it is possible to cluster wrong generators as critical machines.

Based on the results presented in Fig. 13.6 and in order to avoid a wrong selection of critical machines, the previous normalisation of rotor angles suggested in [17] has also been programmed in the SIME implementation. Figure 13.7 presents the time evolution of rotor angles previously normalised. It is possible to see how the critical machines can be now determined easily.

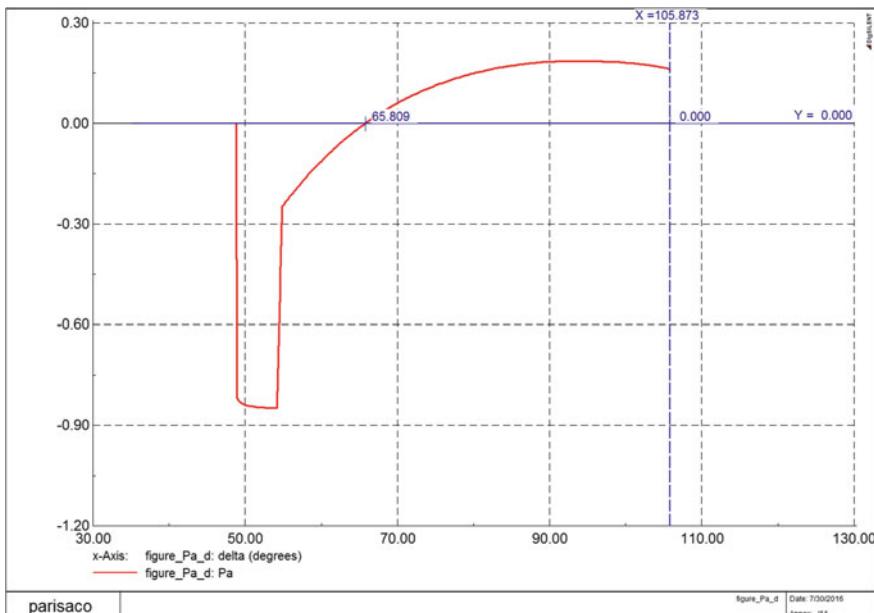


Fig. 13.9 OMIB accelerating power [pu] versus OMIB rotor angle [degrees]

13.4.2.2 Transient Stability Margins

Once the OMIB equivalent has been determined by SIME, the OMIB accelerating power (P_a) and rotor angle (δ) are computed over the time evolution. Figure 13.8 presents the $P_a - \delta$ curve obtained for the analysed unstable study case (note that the ordinate axis is in fact P_a). It is possible to note that the accelerating area is greater than the decelerating area when the OMIB equivalent rotor angle reaches the critical value of 128.5° , approximately at time $t = 601.67$ ms, after which the system will irrevocably reach an unstable condition.

As part of the implemented SIME, as explained in Sect. 13.3, the main results, including the stability margins, are reported in the PowerFactory *Output Window*.

The following report presents these main results for the analysed unstable case. It can be appreciated which generators constitute the critical machines (G8 and G9 in this case), which is the critical time (unstable time), the stability margin (η_u), as well as the angle at which the system instability occurs (unstable angle). A very important result is the instability margin calculation that, in this case, is $\eta_u = -3.075$. One of the greatest advantages found in the application of SIME methodology is that with a less than two-second simulation (in this case less than one second), it is feasible to determine the transient stability margins of a power system.

```
DIGSI/info - DPL Program 'SIME_10G_39B' started
time-step interval simulation (ms)= 220
time-step interval simulation (ms)= 320
time-step interval simulation (ms)= 420
time-step interval simulation (ms)= 520
time-step interval simulation (ms)= 620
Critical Machine 8
Critical Machine 9
UNSTABLE CASE
Pa(pu)= 0.00264646
w_OMIB(pu)= 0.00657816
unstable time tu (ms)= 601.667
unstable angle delta_u (rad)= 2.24278
unstable angle delta_u (degrees)= 128.502
stability margin n_u= -3.07497
Processing time 1.11
DIGSI/info - (t=621:667 ms) DPL program 'SIME_10G_39B' : 'exit'
```

13.4.3 Simulation Results—Stable Case

A three-phase short circuit is applied on line 2–25 at $t = 0.1$ s, followed by the opening of the corresponding transmission line at $t = 0.18$ s (i.e. fault clearing time). The results obtained from the application of the developed SIME DPL are presented as follows.

13.4.3.1 Transient Stability Margins

Figure 13.9 presents the $P_a - \delta$ curve obtained for the analysed stable study case, where it is possible to note that the decelerating area is greater than the accelerating area. Thus, the system is stable, and the stability margin is positive ($\eta_{st} = 0.66$). The report of results corresponding to this stable case shows that G8 and G9 generators are the critical machines. Additionally, it is observed that both the accelerating power (P_a) and the speed (ω) are negative, which implies a stability condition at time $t = 751.67$ ms. At this point, it is determined a recovery angle of 105.87° and a stable and positive stability margin of $\eta_{st} = 0.66$. Note that the developed SIME application allows, through a quadratic regression, knowing the extrapolated unstable angle that in this case is 123.37° .

```
DIgSI/info - (t=781:667 ms) DPL Program 'SIME_10G_39B' started
time-step interval simulation (ms)= 180
time-step interval simulation (ms)= 280
time-step interval simulation (ms)= 380
time-step interval simulation (ms)= 480
time-step interval simulation (ms)= 580
time-step interval simulation (ms)= 680
time-step interval simulation (ms)= 780
Critical Machine 9
Critical Machine 8
STABLE CASE
Pa(pu)= -0.159181
w_OMIB(pu)= -7.1642e-005
recovery time tr (ms)= 751.667
recovery angle delta_r (rad)= 1.84782
recovery angle delta_r (degrees)= 105.873
unstable angle delta_u (rad)= 2.15323
unstable angle delta_u (degrees)= 123.371
stability margin n_st= 0.659003
Processing time (s) 1.73
DIgSI/info - (t=781:667 ms) DPL program 'SIME_10G_39B' : 'exit'
```

13.4.4 Simulation Results—Emergency Control Case

Similarly, to the unstable case presented in Sect. 13.4.2, a three-phase short circuit is applied on line 2–25 at $t = 0.1$ s, followed by the opening of the corresponding transmission line at $t = 0.22$ s (i.e. fault clearing time). However, in the present case, an emergency control action has been implemented for recovering the synchronism of the system. In this connection, it is considered that fast valving control has been installed on the critical machines already determined in the unstable case (G8 and G9). The results obtained from the application of the developed SIME DPL scripts are presented as follows.

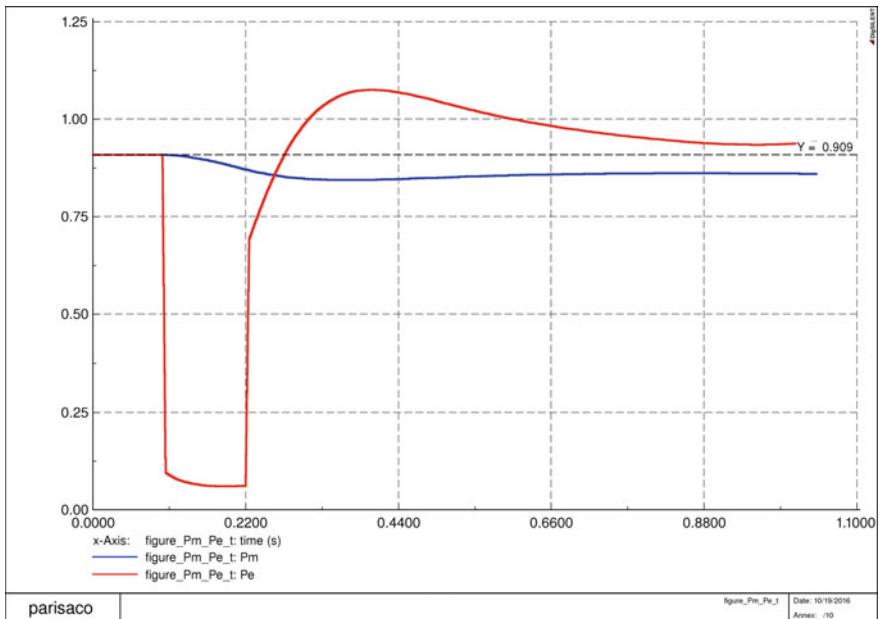


Fig. 13.10 OMIB mechanical and electric power [pu] versus Time [s]

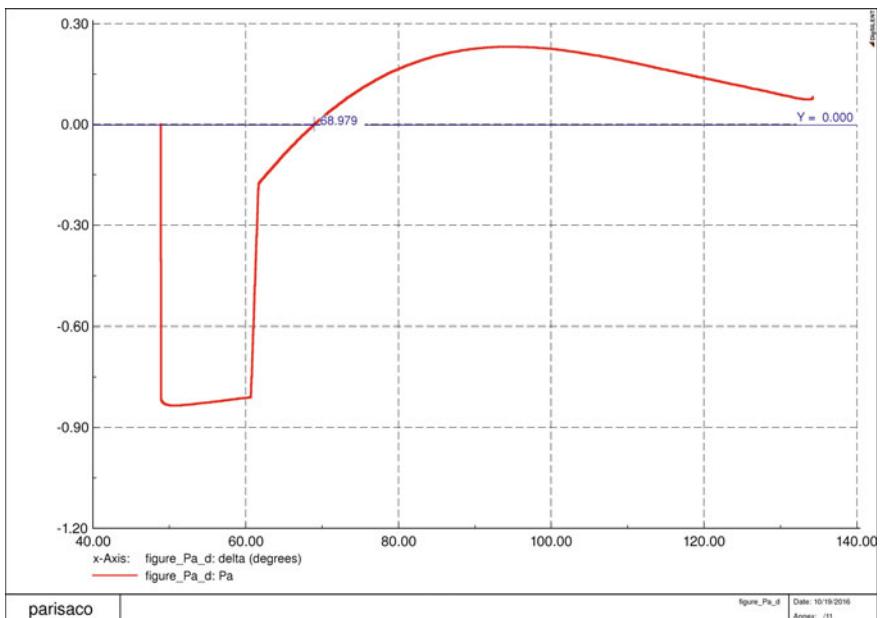


Fig. 13.11 OMIB accelerating power [pu] versus OMIB rotor angle [degrees]

13.4.4.1 Fast Valving Control

- 1 Fig. 13.1.a (unstable case) shows that the excess of the accelerating area over the decelerating area is caused by keeping the high value of mechanical power. Reduction in mechanical power P_m will reduce the accelerating area and enlarge the decelerating area [18]. A proven method to maintain synchronism without generator tripping is the rapid reduction of mechanical power for a defined time period to keep the generator acceleration in an acceptable range; this method is denominated as fast valving [19].
- 2 Fast valving is one of the effective and economical means of improving the stability of a power system under large and sudden disturbances. Fast valving schemes involve rapid closing and opening of steam turbine valves in a prescribed manner to reduce the generator acceleration following a severe fault [20]. For maximum gains with fast valving, the turbine driving power should be reduced as rapidly as possible.
- 3 Fast valving directly acts on the steam valves by omitting the speed turbine control and enables a rapid control of the mechanical power produced by the steam turbine which is submitted to the generator. Thus, fast valving leads to a reduction of the accelerating power [19]. The generators to be controlled by fast valving (i.e. steam units with available fast valving control) should be chosen from the list of unstable generators; in this case, it is assumed that generators G8 and G9 are steam units with available fast valving control.

13.4.4.2 Transient Stability Margins

The power system simulations were performed by SIME DPL scripts to determine the effects of fast valving on the stability of New England power system. For this purpose, the governor TGOV2 (steam turbine governor with fast valving), available at DIgSILENT PowerFactory library, is implemented to the generators G8 and G9.

Figure 13.10 presents the $P_e - t$ and $P_m - t$ curves obtained for this study case, where it is possible to note a significant improvement in the transient stability performance of the power system as regards the unstable case when the fast valving is used in generators G8 and G9. Fast changes in mechanical power demand a fast response of the turbine. In this connection, the time response of steam turbines may be controlled to be as fast as necessary by the fast valving. Figure 13.10 shows how the power system response is considerably improved due to the inclusion of fast valving in the generators G8 and G9. It has been demonstrated how the inclusion of fast valving allows a fast reduction of mechanical power when the generator is losing synchronism, achieving, in fact, the stabilisation of the system in this specific study case.

Figure 13.11 presents the $P_a - \delta$ curve obtained, where it is possible to note that the decelerating area is greater than the accelerating area; therefore, according to the criterion of equal areas, the system is stable.

Table 13.1 Summary of results

Stability assessment results	Unstable case	Stable case	Emergency control case
Stability status	Unstable	Stable	Stable
Critical machines	G8, G9	G8, G9	G8, G9
P_a (pu)	0.0026	-0.1592	-0.0813
ω OMIB (pu)	0.0066	0	0
Unstable time t_u (ms)	601.667	-	-
Recovery time t_r (ms)	-	751.667	1041.67
Unstable angle ($^{\circ}$)	128.502	123.371	138.612
Recovery angle ($^{\circ}$)	-	105.873	134.240
Stability margin	-3.075	0.659	0.0993

Thus, the system is stable, and the stability margin is positive ($\eta_{st} = 0.099$). The report of results corresponding to the fast valving shows that G8 and G9 generators are the critical machines . Additionally, it is observed that both the accelerating power (P_a) and the speed (ω) are negative, which implies a stability condition at time $t = 1041.67$ ms. At this point, it is determined a recovery angle of 134.24° and a stable and positive stability margin of $\eta_{st} = 0.099$. Note that the developed SIME application allows simulating real cases, where it is possible to assess the influence of generators speed governors in the power system.

```
DIGSI/info - DPL Program 'SIME_10G_39B' started
time-step interval simulation (ms)= 220
time-step interval simulation (ms)= 320
time-step interval simulation (ms)= 420
time-step interval simulation (ms)= 520
time-step interval simulation (ms)= 620
time-step interval simulation (ms)= 720
time-step interval simulation (ms)= 820
time-step interval simulation (ms)= 920
time-step interval simulation (ms)= 1020
time-step interval simulation (ms)= 1120
Critical Machine 9
Critical Machine 8
STABLE CASE
Pa(pu)= -0.081291
w_OMIB(pu)= -8.79056e-006
recovery time tr (ms)= 1041.67
recovery angle delta_r (rad)= 2.34293
recovery angle delta_r (degrees)= 134.24
unstable angle delta_u (rad)= 2.41924
unstable angle delta_u (degrees)= 138.612
stability margin n_st= 0.0992598
Processing time (s) 3.06
DIGSI/info - (t=01:122 s) DPL program 'SIME_10G_39B' : 'exit'
```

13.4.5 Summary of Results

In this section, three different study cases have been presented. The results obtained from the different simulations have allowed highlighting the effectiveness of the implemented SIME method for assessing the transient stability of a power system modelled in DIgSILENT PowerFactory.

Table 13.1 presents a summary of the stability assessment results from the three simulated cases. It is possible to appreciate how the magnitude and sign of the stability margin give an important indicator of the system stability level, being the more negative values the worst stability cases.

13.5 Concluding Remarks

This chapter presents the development of a computational tool based on the single machine equivalent (SIME) method, which assesses transient stability problems of a power system through stability margins. This evaluation allows guiding the definition of preventive control actions in order to improve power system dynamic security. Furthermore, this tool can help users to structure smart grid applications, even those regarding real-time assessment and corrective control actions (emergency single machine equivalent E-SIME).

Using the DIgSILENT Programming Language (DPL), a computational tool for evaluating the transient stability and obtaining stability margins based on SIME method was developed. This application evaluates the behaviour of the power system in its post-fault configuration in terms of equivalent one machine infinite bus (OMIB). The evolution of mechanical and electrical parameters (rotor angle, rotor speed, mechanical and electric power) of OMIB is calculated from the evolution over time of the electrical quantities of the machines and from the equal area criterion EAC. It allows calculating transient stability margins of the power system.

The results obtained by implementation and application of the methodology in the test system IEEE New England allowed verifying the benefits of the method for calculating transient stability margins. Simulations have been run in Windows 7 Enterprise-Intel Core i7-4700MQ CPU @2.4 GHz 2.4 Ghz 8 GB RAM. Under this ambient, the resulted SIME computation elapsed time, for a large size power system (Ecuadorian electric power system: 165 generators and 561 buses), resulted in an average of 5 s faster than the common ten-second window time-domain simulation. Another important aspect to note is that the time-domain simulations only provide qualitative results that are dependent on the user's point of view. Thus, in order to be sure that a power system is unstable, several time-domain simulations might be necessary to be performed even with different time windows. This aspect might increase even more the computation time.

Following this same research line, future work is being developed in order to incorporate E-SIME concepts into the *SIME_10G_39B.ComDpl* script (e.g. the prediction of angle trajectories) with the aim of analysing the real-time transient stability assessment problem.

References

1. G. Andersson, P. Donalek, I. Kamwa, P. Kundur, et al., Causes of the 2003 Major Grid Blackouts in North America and Europe, and recommended means to improve system dynamic performance. *IEEE Trans. Power Syst.* **20**, N° 4, Nov 2005
2. M. Amin, Toward self-healing infrastructure systems, Electric Power Research Institute (EPRI), IEEE, 2000
3. J. Cepeda, *Real Time Vulnerability Assessment of Electric Power Systems using Synchronized Phasor Measurement Technology*, Ph.D. Thesis, Instituto de Energía Eléctrica, Universidad Nacional de San Juan, San Juan, Argentina, Dec 2013
4. P. Kundur, J. Paserba, V. Ajjarapu et al., Definition and classification of power system stability IEEE/CIGRE joint task force on stability terms and definitions. *IEEE Trans. Power Syst.* **19**(3), 1387–1401 (2004)
5. D. Echeverría, J. Rueda, G. Colomé, I. Erlich, Improved method for Real-Time Transient Stability Assessment of Power Systems, in *Proceedings of the IEEE PES General Meeting*, San Diego, California, July 2012
6. J. Cepeda, G. Colomé, N. Castrillón, Dynamic Vulnerability Assessment due to Transient Instability based on Data Mining Analysis for Smart Grid Applications, in *Proceedings of the IEEE PES ISGT-LA Conference*, Medellín, Colombia, Oct 2011
7. S. Savulescu et al, Real-Time Stability Assessment in Modern Power System Control Centers (IEEE Press Series on Power Engineering 2009), 2009
8. Z. Huang, P. Zhang et al., Vulnerability Assessment for Cascading Failures in Electric Power Systems, Task Force on Cascading Failures, in *Proceedings of the IEEE PES Power Systems Conference and Exposition*, Seattle, 2009
9. J. Cepeda, J. Rueda, G. Colomé, D. Echeverría, Real-time Transient Stability Assessment Based on Centre-of-Inertia Estimation from PMU Measurements. *IET Gener. Trans. Distrib.* **8** (8), Aug 2014
10. Y. Zhang, P. Rousseaux, I. Wehenkel, M. Pavella, SIME: A Comprehensive Approach to Fast Transient Stability Assessment, in *Proceedings of IEE-Japan, Power and Energy '96*, (Osaka, Japan, 1996), pp. 177–182
11. M. Crappe, *Electric Power Systems* (Wiley, ISTE Ltd, 2008)
12. M. Eremia, M. Shahidehpour, *Handbook of Electrical Power System Dynamics* (IEEE Press, Wiley, United States of America, 2013)
13. Y. Zhang, L. Wenhenkel, M. Pavella, SIME: A hybrid approach to fast transient stability assessment and contingency selection. *Electr. Power Energy Syst.* **19**(3), 195–208 (1997)
14. M. Pavella, D. Ernst, D. Ruiz-Vega, *Transient Stability of Power Systems: A Unified Approach to Assessment and Control* (Kluwer, Norwell, 2000)
15. P. Hirsch, D. Sobajic, Single Machine Equivalent (SIME) Approach to Dynamic Security Assessment (DSA), EPRI, Nov 2000
16. DIgSILENT GmbH, DIgSILENT PowerFactory Version 15 User's Manual, Gomaringen, Germany, July 2014

17. D. Echeverria, J. Cepeda and D. Colomé, Critical machine identification for power systems transient stability problems using data mining, in *IEEE PES Transmission & Distribution Conference and Exposition—Latin America (PES T&D-LA)*, Medellin, 2014
18. J. Machowski, A. Smolarczyk, J. W. Bialek, Power System Transient Stability Enhancement by Co-ordinated Fast Valving and Excitation Control Of Synchronous Generators, in *CIGRE Symposium “Working Plants and Systems Harder*, London, June 1999, paper 200-06, pp. 7–9
19. I. Erlich, J. Löwen, J. Schmidt, W. Winter, Advanced requirements for thermal power plants for system stability in case of high wind power infeed, in *7th International Workshop on Large Scale Integration of Wind Power and on Transmission Networks for Offshore Wind Farms*, Madrid, Spain, 2008
20. P. Kundur, *Power System Stability and Control*, McGraw-Hill, Inc., 1994

Chapter 14

Generic DSL-Based Modeling and Control of Wind Turbine Type 4 for EMT Simulations in DIgSILENT PowerFactory



**Abdul W. Korai, Elyas Rakhshani, José Luis Rueda Torres
and István Erlich**

Abstract In this chapter, to cope with new challenges arising from the increasing level of power injected into the network through converter interfaces, a new wind turbine (WT) as well as a VSC–HVDC control concept, which determines the converter reference voltage directly without the need for an underlying current controller, is presented and discussed. Additionally, alternative options for frequency support by the HVDC terminals that can be incorporated into the active power control channel are presented. The implementation steps performed by using DSL programming are presented for the case of EMT simulations. Simulation results show that the control approach fulfills all the operational control functions in steady state and in contingency situations supporting fault ride through and emergency frequency support, without encountering the problems arising from current injection control.

Keywords DIgSILENT EMTP simulation · DSL programming
Wind turbine control · HVDC control

Electronic supplementary material

The online version of this chapter (https://doi.org/10.1007/978-3-319-50532-9_14) contains supplementary material, which is available to authorized users.

The original version of this chapter was revised: ESM files have been included. The erratum to this chapter is available at https://doi.org/10.1007/978-3-319-50532-9_15

A. W. Korai · I. Erlich
Institute of Electrical Power Systems, University Duisburg-Essen,
Duisburg, Germany

E. Rakhshani · J. L. Rueda Torres (✉)
Department of Electrical Sustainable Energy, Delft University
of Technology, Delft, Netherlands
e-mail: j.l.ruedatorres@tudelft.nl

14.1 Introduction

Application of proper software, like DIgSILENT PowerFactory, for analyzing the future grids with power electronic-based generation like wind turbine (WT) generation considering the fact that future WT will be required to participate on active and reactive power management systems, is the motivation of presented material in this chapter. Future WT should be able to contribute on reactive power management and be involved in continuous voltage control like the synchronous machines currently are. Furthermore, how and in what form the converter-based generations may be involved in primary frequency control or even providing virtual inertia is still a challenging issue.

This chapter provides a control approach which achieves all the standard control objectives without encountering the problems arising from current injection control from the conventional control of WT. On the basis of these new control approaches, the implementation of coordinated var-voltage control and alternative frequency support schemes will be presented and simulation results that demonstrate the feasibility of the new approach will be provided. All the simulation are presented and discussed using the functionalities of DIgSILENT simulation language (DSL) for EMT-based models.

This chapter is organized as follows: In Sect. 14.2, a review on wind model and its mathematical equations is presented briefly while in Sect. 14.3 the background on the current injection-based control for WT is reviewed. Then, in Sect. 14.4, details of the proposed generic control approach for WT is presented and discussed. In Sect. 14.5, DSL-based control implementation in DIgSILENT PowerFactory is explained and finally in Sect. 14.6 the performance of the proposed control for WT modeling in DIgSILENT is simulated and analyzed for one power system case study with different contingencies. Summary and main conclusions of this chapter are also presented in Sect. 14.7.

14.2 Wind Turbines Technologies

There are four main technologies for wind turbine (WT), such as fixed-speed, variable speed, doubly fed induction generator (DFIG)-based WT and variable speed full converter wind turbine (FCWT). The first concept is a fixed-speed wind turbine equipped with a squirrel cage induction generator. The second one is a variable speed wind turbine with variable rotor resistance, which usually is used by Vestas [1]. This type of wind turbine technologies is using a wound rotor induction generator which is equipped with variable rotor resistances. The rotor resistances are regulated by means of a power converter. This concept is known as the limited variable speed concept. The third technology for WT is based on a variable speed wind turbine with a partial power converter or a wind turbine with a doubly fed induction generator (DFIG).

The fourth technology will be a variable speed FCWT. This concept can use different type of generators, like induction generator or a synchronous generator, with permanent magnets or external electrical excitation.

In case of FCWT, its functionality is more adaptable with the stringent grid codes which are currently required wind turbines to have high immunity against grid faults. In addition, FCWT type wind turbines can provide more reactive power support which are highly preferable. For these reasons, FCWTs are increasingly penetrating the market.

Normally, the turbine concept will be consisted of one generator which is interfaced directly to the grid with a back to back converter. This generator can be a synchronous generator, with or without a gearbox, or an induction generator type including its gearbox. The converter part is normally made of IGBTs [2] while the grid/line-side converter, which mainly acts as a current-regulated, voltage-fed inverter, can provide reactive power support to the grid. In case of FCWT, different types of generators can be used like a permanent magnet synchronous generator (PMSG), an electrically excited synchronous generator and an induction generator. As it is explained, modern wind turbines are able to control active and reactive current and thus active and reactive power independently. In the wind turbine concept with a fully rated converter (so-called Type 4 wind turbine, Fig. 14.1), the grid or line-side converter (LSC) is fully responsible for the control actions toward the power system and rotor-side converter (RSC) is responsible for converter control actions [3, 4]. Therefore, the LSC is usually equipped with decoupled current control in which the active power output is controlled via the DC voltage and the reactive power, alternatively power factor or terminal voltage, is regulated via other external control loops.

In steady state, the priority of the LSC control is on the feed in of active power delivered by the wind turbine. However, the wind turbines are also able to supply reactive power within a certain range in addition to the active power.

In case of large voltage drop or voltage rise, converter switches from active current priority to a reactive current priority. This means that the active current and thus active power will be reduced, if necessary, in favor of reactive current. The aim of injecting capacitive or inductive current during large voltage deviations is the stabilization of the voltage [5].

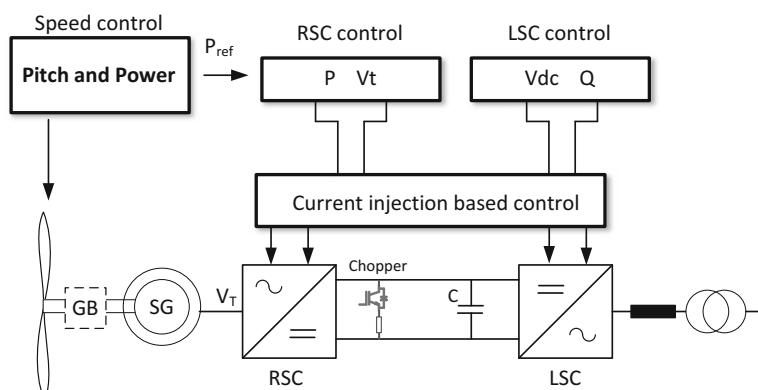


Fig. 14.1 Type 4 full converter wind turbine

14.2.1 Full Power Converter Wind Turbine with PMSG

Since in a permanent magnet synchronous generator the excitation can be provided by the permanent magnets, it does not require a separate rotor excitation. Thus, no rotor winding is needed. This advantage can significantly reduce the excitation losses as well as the torque density of the generator [6].

In the PMSG model, it is assumed that the flux is sinusoidally distributed along the air gap and therefore, no damping winding is considered. The mathematical equations can be made by aligning the d -component of machine vectors to the rotor flux. Therefore, voltage equations of the machine will be as follows:

$$v_{sd}^r = R_s i_{sd}^r - \omega_r \psi_{sq}^r + \frac{d\psi_{sd}^r}{dt} \quad (14.1)$$

$$v_{sq}^r = R_s i_{sq}^r - \omega_r \psi_{sd}^r + \frac{d\psi_{sq}^r}{dt} \quad (14.2)$$

with the stator flux components obtained from

$$\psi_{sd}^r = X_{sd} i_{sd}^r + \psi_{pm}^r \quad (14.3)$$

$$\psi_{sq}^r = X_{sq} i_{sq}^r \quad (14.4)$$

where v_{sd}^r and v_{sq}^r are the d and q components of the terminal voltage vector, respectively; i_{sd}^r and i_{sq}^r are the d and q components of the stator currents, respectively; R_s is the stator resistance; X_{sd} and X_{sq} are the d and q components of the stator reactances, respectively, and ψ_{pm}^r is the permanent magnet flux linkage. Superscript r will indicate the corresponding variable which is related to the rotor flux reference frame.

The electromagnetic torque of PMSG can be also expressed by

$$T_e = i_{sq}^r \left(i_{sd}^r (X_{sd} - X_{sq}) + \psi_{pm}^r \right) \quad (14.5)$$

For a non-salient pole machine, the stator inductances X_{sd} and X_{sq} can be assumed to be equal. Consequently, the d -component of the stator current i_{sd}^r does not influence the electromagnetic torque.

14.2.1.1 Rotor-Side Controller

Typically, the generator/rotor-side converter will regulate generator speed and power [7, 8]. The converter can be also employed to control the dc-link voltage [2]. In addition to these control actions, the converter can be used to control the reactive power exchange with the generator. The controller can be implemented using the

vector control technique, in a way that the d -axis is aligned to the rotating stator flux. Using this control method, various control strategies like full torque control or constant stator voltage can be implemented.

14.2.1.2 DC-link and Line-Side Controller

The line-side converter equations can be expressed in dq -components aligned with the grid voltage vector. Normally, the line-side converter is assigned to maintain the dc-link voltage level and reactive power injection into the grid [7, 8]. Otherwise, the line-side converter can also be used to control the active power of the wind turbine. An independent control of active power or the dc-link voltage on one side, and reactive power on the other side, can be realized using a vector control aligned with the grid voltage vector.

Finally, the complete model of WT will have several order but sometimes it is possible to reduce the order of such model or make some simplification depending on the objectives of the analysis. Complete details of turbine modeling and mechanical components can be found in [4]. For further simplification, mechanical equations can be also omitted and DC input can be considered with a controlled current source. It should be mentioned that the aerodynamic model is, in fact, the same as those in wind turbines with doubly fed-induction generators as described in [4]. The main purpose of the pitch controller is to limit generator speed and power output at rated value.

14.3 Review on Current Injection-Based Control of WT

The common current control approach shown in Fig. 14.2 uses two current feed-forward terms and two PI control blocks. Furthermore, the grid voltage is added to the d -channel output which represents an additional feed-forward term.

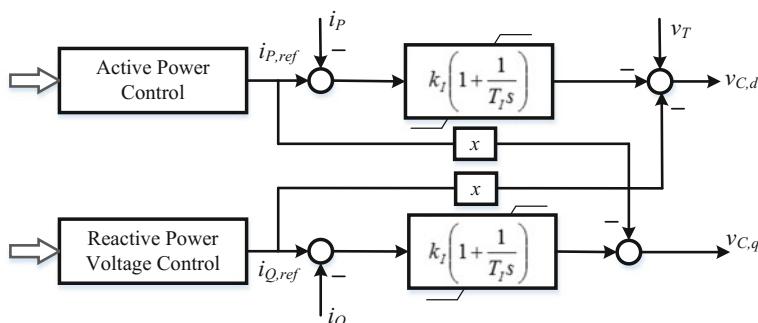


Fig. 14.2 Classical control approach

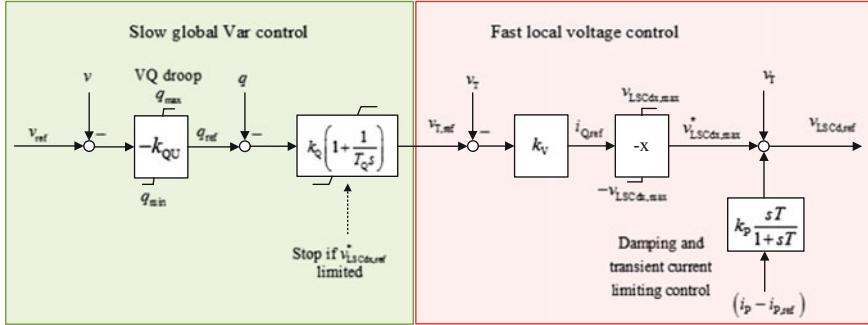


Fig. 14.3 Var-voltage control approach for WT

Control input of the active current controller is the active current reference forwarded from the DC voltage control.

Feedback signal is the measured active current. Control input for the reactive current controller is the reactive current set point from outer reactive current set-point calculation (see Fig. 14.3). The feedback is the measured reactive current where k_I and T_I are controller parameters which act as a filter, i_P and i_Q denote the grid current components, v_T is the grid voltage, $v_{C,d}$ and $v_{C,q}$ denote the converter voltage vector components.

The output of both PI controllers is ideally zero because the feed-forward terms take care of the main control actions. The PI controllers account for parameter and measurement uncertainties and are normally limited to the nominal current of the LSC. Output of the current control is the reference voltage of the converter [4].

With the projected future scenarios ranging from 50 to 100% renewables, it is obvious that a new converter control scheme is required which adapts automatically to the current depending on the grid conditions by trying to fulfill the main objectives of voltage and frequency control.

14.4 Proposed Generic Control Scheme for WT Application

In this section, a generic control technique for grid forming WTs is proposed. In the presented control for WT, on the one hand, it does not necessitate current injection for the controller to function properly, and on the other hand, it is capable of mitigating some of the problems arising from the presence of converters in large number.

The main control loops of the presented control approach are var-voltage and active power/frequency control loops.

14.4.1 Var-voltage Control Channel

As shown in Fig. 14.3, the reactive power reference, in the proposed solution for the var-voltage control channel, is based on a droop characteristic. The response time of this controller is set 5–30 s, large enough to avoid significant control action during network short circuit and small enough to preclude unnecessary tap movement in on-load tap changing (OLTC) transformers.

This control approach consists of a slow global controller, as explained, and a fast local controller for voltage control and damping.

The damping term replaces the proportional component of the PI block in the standard implementation at the moment. It offers also additional possibilities in terms of selective damping, and for the converter required current limitation is achieved through output voltage limitation. After the addition of the controlled terminal voltage (the feed-forward term), the d-component of the inverter voltage is determined. As can be seen, the reactive current no longer appears explicitly in this scheme, and as a result, there is no risk of integrator windup in the event of an unforeseen islanding. The current adjusts itself according to the network conditions in response to a changing converter voltage. The output of the voltage controller is also limited for taking into account the current limitation of the converter. The maximum value $v_{LSC,max}$ was calculated depending on the selected active or reactive current priority. The limited output is then added to the measured grid voltage (see Fig. 14.3). The result represents the LSC voltage reference in the d-axis. This maximum value calculation of those limiter can be also applied for active power controller of the next subsection.

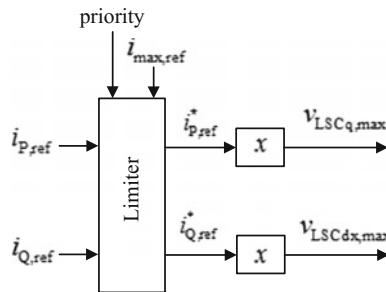
The converter current is limited through the voltage thresholds $v_{LSCdx,max}$ and $v_{LSCq,max}$. Calculations can be made as follows:

14.4.1.1 Part 1: Limiting of the Real Current (Current Limiting Control)

If $(|i_d + j i_d| - i_{max0_ref}) > 0 \rightarrow i_{max_ref} = i_{max0_ref} - k_{red} \cdot (|i_d + j i_d| - i_{max0_ref})$
else $i_{max_ref} = i_{max0_ref}$

14.4.1.2 Part 2: Limiting of the Current Reference

As long as the reference current is not limited $v_{LSCd,max} = v_{LSCq,max} = x \cdot i_{max,ref}$
otherwise $v_{LSCd,max} = x \cdot i_Q^*, ref$; $v_{LSCq,max} = x \cdot i_P^*, ref$



Considering current priority in reference current limitation:

14.4.2 Active Power Control Channel

The proposed control schema is shown in Fig. 14.4. The structure of DC voltage controller with the PI characteristic is the same as in previous controller schemes. The output of the DC voltage controller is the active power injected into the network:

$$p = v_t i_p = -v_t \frac{v_{cq}}{x} \quad (14.6)$$

As can be seen in (14.6), the active power can be controlled using the q-component of the converter voltage, which can also be used to limit the current. Please note that no integral current injection is used, and the actual active current adjusts itself in accordance with the power flow equations of the network. For frequency control, three options are given in Fig. 14.5. As it has been explained, the current magnitude limitation is also possible according to the selected priority.

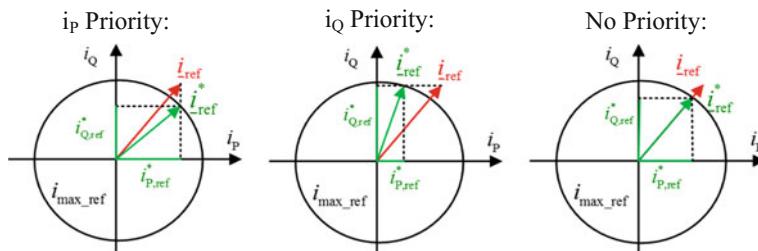


Fig. 14.4 Current priority limitations

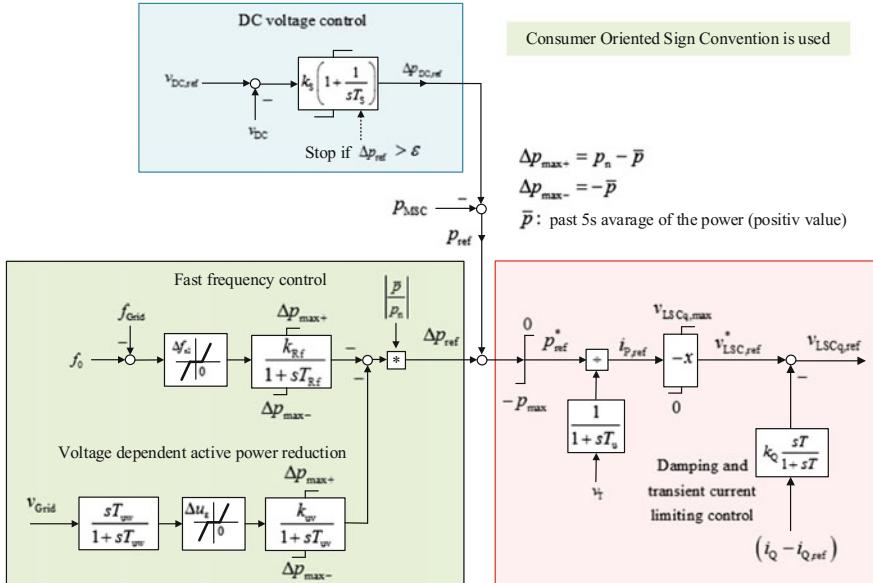


Fig. 14.5 Proposed control schema in active power control channel

14.4.2.1 Option A: Over-frequency Emergency Control (OFEC)

The control is activated when the frequency exceeds a preset threshold value, e.g., 50.2 Hz in the example in this paper. The gain k_{RF} defines the frequency deviation at which the power reduction corresponds to the total power p_{ref} (e.g., 51.5 Hz). The time constant T_{RF} is small totally with the fast response time of converters. One can also define in the delay block a limitation of rate of change, the effect of which will be demonstrated in the next section.

14.4.2.2 Option B: Direct Frequency Control

This method is similar to Option A with the only difference that frequency control takes place for frequency errors in both directions, i.e., at both under frequency as well as over frequency. Normally, this control is significantly slower than Option A and contains only a small dead band (e.g., 20 MHz).

It is intended to contribute to frequency regulation in normal operation and is also a requirement in ENTSO-E.

14.4.2.3 Option C: Inertia Control

This option represents the well-known inertia control with extension in both frequency change directions. This means the controller can also contribute to the stabilization of the frequency by reducing the power at the expense of a speed increase in wind turbines during an over-frequency event. In the case of grid frequency drop, the controller will increase the power temporarily. The activity of this controller is restricted to a narrow time band due to the washout filter at the frequency input.

14.5 DSL-Based Control in DIgSILENT PowerFactory

Figure 14.6 shows the model of the grid side converter, which was built as a composite model and contains the following components: (a) converter used in wind farm model which includes the PWM converter (ElmVsc), (b) DC voltage and power controller (Udc_PQ controller, ElmDsl); (c) current controller (ElmDsl), (d) DC model (ElmDsl), (e) AC voltage measurement (StaVmea), (f) AC current measurement (StaImea), (g) power calculation block (ElmDsl), (h) Park transform (ElmDsl), model of current source used in simplified WT (ElmDsl), PLL: angle measurement (ElmPhi_pll), and (i) Mod Limiter (ElmDsl).

Correct initialization of a model in a power system simulation tool avoids fictitious electrical transients and makes it possible to evaluate correctly the real dynamic performance of the system. Therefore, the initialization equations of important dynamic block of this composite model are presented below:

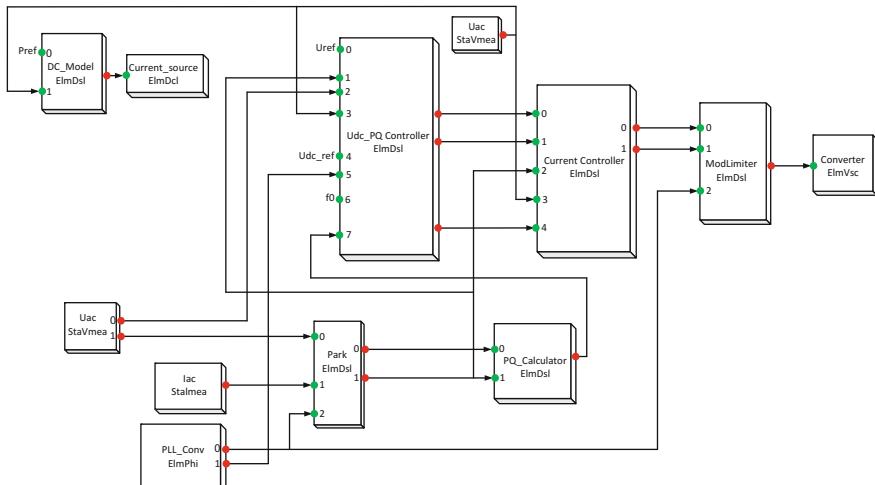


Fig. 14.6 Structure of the grid side converter model in PowerFactory

DC Model initialization:

```
vardef(VdcN)='kV';'Nominal DC Voltage'
vardef(Vdcmax)='kV';'Max DC Voltage'
vardef(Pnom)='MW';'Nominal active power of the converter'

inc(xr)=1
inc(vdc)=1
inc(pref)=-1
```

Mod Limitation initialization:

```
inc(ud_ref)= ur*cos(phi) + ui*sin(phi)
inc(uq_ref)=-ur*sin(phi) + ui*cos(phi)
inc(ur)=Pmr /0.612
inc(ui)=Pmi /0.612
```

Current controller initialization:

```
vardef(Kw) = '-';'Washout out filter gain'
vardef(Tw) = 's';'Washout filter time constant'
vardef(Un) = 'V' ; 'Nominal AC voltage of converter'
vardef(UDCn) = 'V' ; 'Nominal DC voltage of converter'
vardef(ud_min) = 'pu' ; 'Minimum modulation index d-channel'
vardef(uq_min) = 'pu' ; 'Minimum modulation index q-channel'
vardef(ud_max) = 'pu' ; 'Maximum modulation index d-channel'
vardef(uq_max) = 'pu' ; 'Maximum modulation index q-channel'

inc(uq_set)=0
inc(xdw) =0
inc(xqw) =0
inc(ulsc_d)=1

inc(id_ref_)=id
inc(iq_ref)=iq
Ksin = 2*sqrt(2)*Un/(sqrt(3)*UDCn)
```

Udc_PQ Controller initialization:

```

vardef(Kv) = '-' ; 'Global Var control P.constant'
vardef(Tv) = 's' ; 'Global Var control Integral time constant'
vardef(Kdc) = '-' ; 'DC link control P.constant'
vardef(Tdc) = 's' ; 'DC link control Integral time constant'
vardef(uDC_CHon) = 'pu' ; 'Chopper activation voltage level'
vardef(uDC_CHoff) = 'pu' ; 'Chopper deactivation voltage level'
vardef(iChopper) = '-' ; 'Chopper active [1]- inactive[0]'
vardef(Ku) = 'pu' ; 'Fast voltage control P.constant'
vardef(db_spennung_VDAPR) = 'pu' ; 'deadband for VDAPR'
vardef(irmax) = 'pu' ; 'Maximum converter current'
vardef(Tuu) = 's' ; 'Voltage measurement delay'
vardef(T_rate_limit_freq) = 's' ; 'Frequency control time delay'
vardef(l) = 'pu' ; 'Converter reactance'
vardef(T) = 's' ; 'VDAPR washout filter time constant'
vardef(qmax) = 'pu' ; 'Maximum reactive power in steady state'
vardef(Kvdapr) = '-' ; 'Gain of the VDAPR'
vardef(Tvdapr) = 's' ; 'Time delay of the PT1 filter of VDAPR'
vardef(Tp_aver) = 's' ; 'Time constant for the calcualtion of p average'
vardef(Kqu) = '-' ; 'Static gain of the reactive power control'

inc(ulsc_d)=1
inc(xv)= 1.01
inc(qref)=q
inc(xu)=ulsc_d
inc(id_ref)=-0.95/us
inc(idref)=-0.95/us
inc(uref)=us
inc(xdc)=id
inc(udc_ref)=1
inc(xst)= -0.95
inc(xuuu)=0
inc(xuu)=us
inc(x)=us
inc(f0)=50
inc(xp_avg)=p
inc(xf)=0

```

14.6 Performance Evaluation in DIgSILENT PowerFactory

As shown in Fig. 14.7, wind farm model used in this chapter is a simplified model including controlled current source, wind IGBT-based converter, chopper, series reactor and compensation and grid side transformer.

An example of a grid model is also depicted in Fig. 14.8. It contains the connection of the wind farms with 25 MW and five units of 5 MW farms to the main grid. The connection of the wind turbine to the station is modeled by the actual physical component models from DIgSILENT library (transformer, line, load, bus bar), while the remaining power system is represented by simplified equivalents, as, e.g., a Thevenin equivalent models the grid. This is a reasonable approximation for

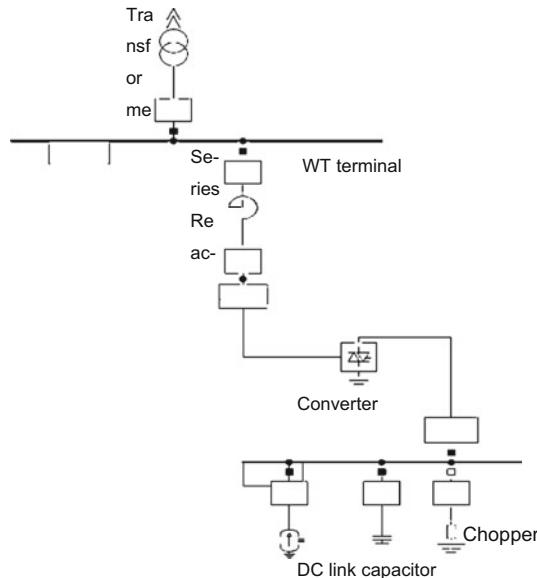


Fig. 14.7 Single line diagram of the full converter model in DIgSILENT

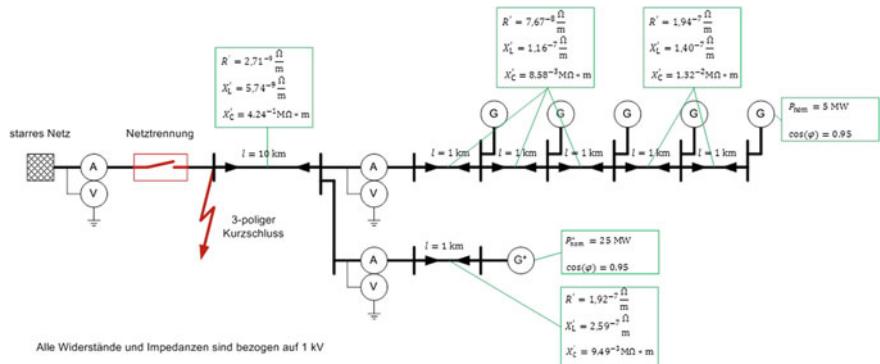


Fig. 14.8 Test system used for simulations with wind farms

power quality studies, as the grid is assumed very strong as compared to the power capacity of the wind turbine.

Three-phase short circuit and islanding action are used for testing the performance of the proposed model. The time response in terms of grid frequency, bus voltage magnitude as well as injected currents to the grid (i.e., the point of common coupling of the FSC-based WT) for a three-phase grid fault for a duration of 150 ms is analyzed for illustrative purposes.

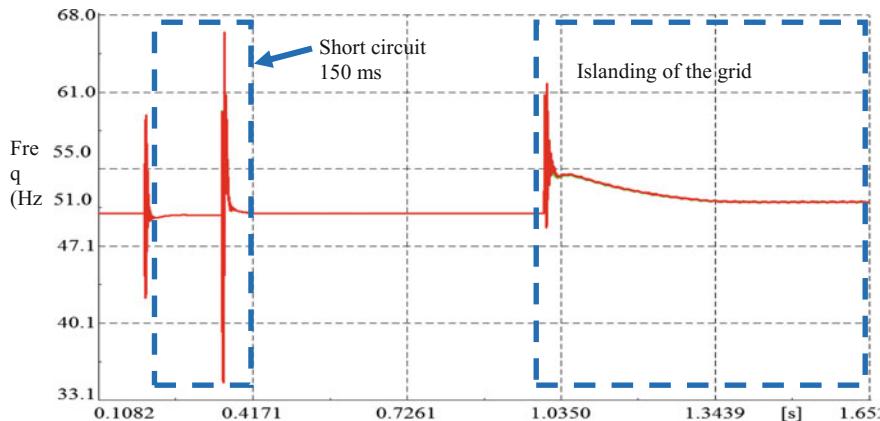


Fig. 14.9 Frequency of the grid

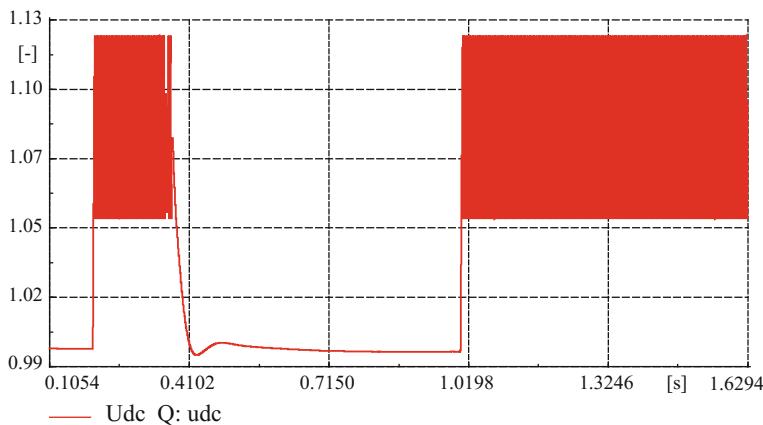


Fig. 14.10 DC-link voltage of the WT

Figures 14.9 and 14.10 show the variation of the frequency of the grid and the DC-link response following the interruption of the connection to the external grid for these scenarios.

Current response of the converter, comparing the measured and reference currents, during three-phase short circuit and for islanding action is also presented in Figs. 14.11 and 14.12, respectively.

Instantaneous voltages and currents are plotted for both the short circuit and islanding events in Fig. 14.13.

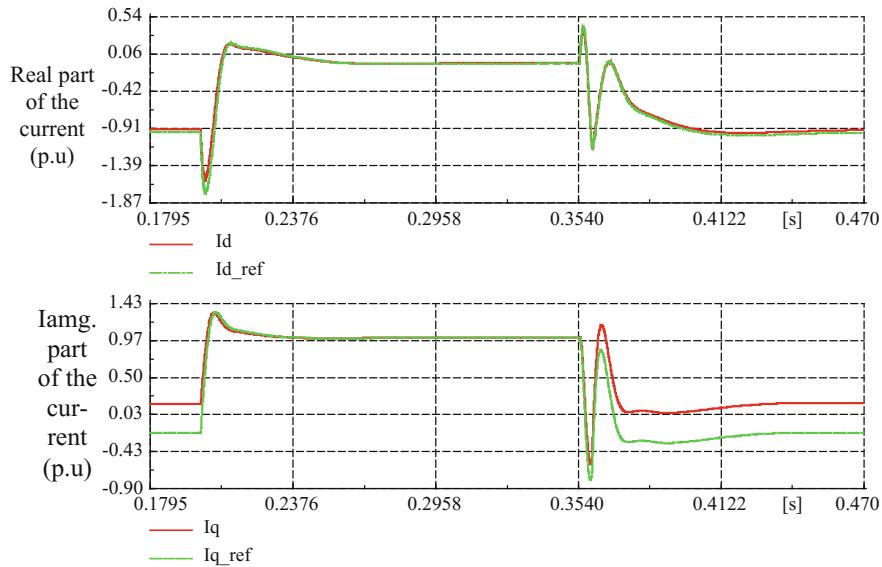


Fig. 14.11 Converter current during three-phase short circuit

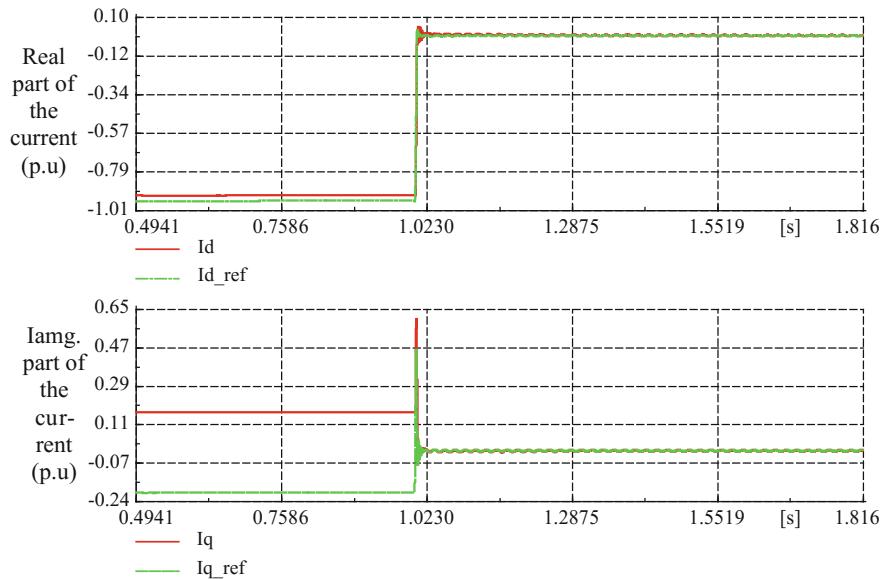


Fig. 14.12 Converter current after islanding

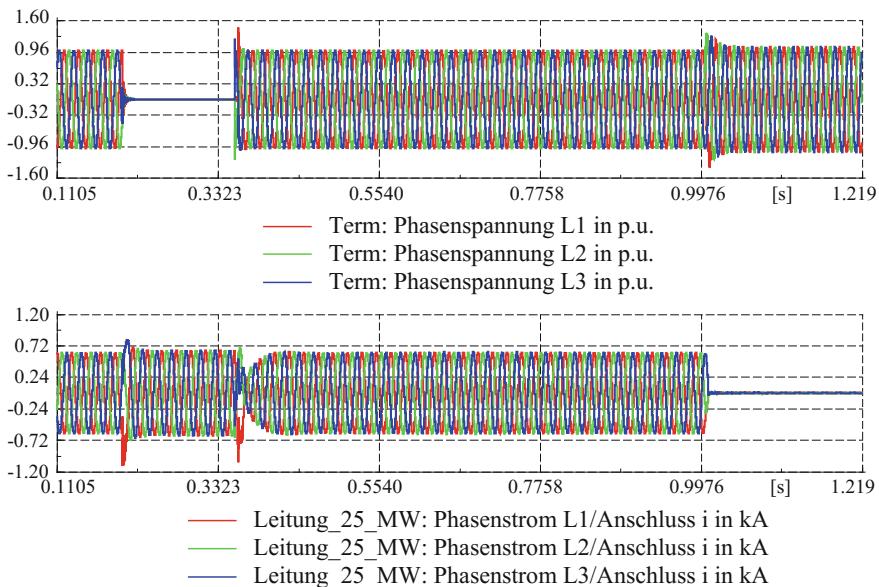


Fig. 14.13 Instantaneous voltages and currents (1 Mvar capacitive, three-phase short circuit at 0.2 s, Islanding at 1 s)

14.7 Conclusions

In this chapter, the implementation of generic model of WT type 4 for dynamic simulations in DIgSILENT PowerFactory was presented. The control implementation and its initialization in PowerFactory were also explained.

Due to the problems raised by classical PI blocks in the current injection-based controller, a new generic control considering both active power and var-Q control is performed. Active power control is performed using the DC voltage controller. For var-voltage control, a hierarchical scheme is suggested based on a slow var controller with PI characteristic and a local fast voltage controller with proportional characteristic. To achieve sufficient damping, two damping control blocks are included to the output of the controllers. These controllers are only active in the transient period due to the wash-out filter implemented.

References

1. General Specification V80—1.8 MW 60 Hz OptiSlipr—Wind Turbine, Vestas, May 2002
2. A. Hansen, F. Iov, F. Blaabjerg, L. Hansen, Review of contemporary wind turbine concepts and their market penetration. Wind Eng. **28**(3), 247–263 (2004)

3. J. Fortmann, S. Engelhardt, J. Kretschmann, C. Feltes, I. Erlich, New Generic Model of DFG-Based Wind Turbines for RMS-Type Simulation. *IEEE Trans. Energy Convers.* **29**(1) (2014)
4. J.L. Rueda Torres, F. Gonzalez-Longatt, *PowerFactory. Applications for Power System Analysis* (Springer, Switzerland, 2015)
5. I. Erlich, B. Paz, M. Koochack Zadeh, S. Vogt, C. Buchhagen, C. Rauscher, A. Menze, J. Jung, Overvoltage phenomena in offshore wind farms following blocking of the HVDC converter, in *IEEE PES General Meeting*, 2016
6. J. Gieras, M. Wing, *Permanent Magnet Motor Technology: Design and Applications* (CRC Press, Boca Raton, FL, 2002)
7. M. Chinchilla, S. Arnaltes, J. Burgos, Control of permanent-magnet generators applied to variable-speed wind energy systems connected to the grid. *IEEE Trans. Energy Convers.* **21**(1), 130–135 (2006)
8. J. Conroy, R. Watson, Low-voltage ride-through of a full converter wind turbine with permanent magnet generator. *IET Renew. Power Gener.* **1**(3), 182–189 (2007)

Erratum to: Advanced Smart Grid Functionalities Based on PowerFactory



Francisco Gonzalez-Longatt and José Luis Rueda Torres

Erratum to:

**F. Gonzalez-Longatt and J. L. Rueda Torres (eds.),
*Advanced Smart Grid Functionalities Based
on PowerFactory, Green Energy and Technology,*
<https://doi.org/10.1007/978-3-319-50532-9>**

The original version of the book was inadvertently published without the ESM files in all the chapters except chapter 1, which have been now included. The erratum book has been updated with the changes.

The updated online version for this book can be found at
<https://doi.org/10.1007/978-3-319-50532-9>