SYNTHETIC DATA-BASED MACHINE LEARNING APPLICATIONS FOR 21ST-CENTURY POWER SYSTEMS

Sergio A. Dorado-Rojas

Submitted in Partial Fullfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Approved by:
Luigi Vanfretti, Ph.D., Chair
Santiago Paternain, Ph.D.
Agung Julius, Ph.D.
Fotios Kopsaftopoulos, Ph.D.

Department of Electrical, Computer, and Systems Engineering
Rensselaer Polytechnic Institute
Troy, New York

[August 2022]
Submitted August 2022

CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

simulation time (ST) the time it takes for a program to translate, compile and perform numerical integration on a Modelica model for a dynamic simulation.

execution time (ET) the elapsed time to complete the numerical integration process of a compiled model. It also known as *integration time*. Note that execution time is included as a part of simulation time (ST) in the context of this document.

normalized minimum execution time (NMT) performance metric taken as a function of the execution time of each of the tools. It is defined as

$$\mathrm{NMT}^{[\mathrm{solver}]} = \frac{\min(\mathrm{ET}_{\mathrm{D}}, \mathrm{ET}_{\mathrm{OM}})}{\mathrm{ET}_{\mathrm{observed}}}$$

where ET_{D} and $\mathrm{ET}_{\mathrm{OM}}$ are the execution times for Dymola and OpenModelica (OM), respectively, and $\mathrm{ET}_{\mathrm{observed}}$ is the corresponding integration time of each tool for a given solver obtained from the simulation log. Note that execution time is included as a part of ST in the context of this chapter.

NOMENCLATURE

OpenIPSL	Open-Instance Power System Library
OM	OpenModelica
MSE	Mean Squared Error
NAE	Normalized Absolute Error
OMPython	OpenModelica-Python Interface
PDI	Python-Dymola Interface
DAE	Differential Algebraic Equation
PSS®E	Power System Simulator for Engineering
NR	Newton-Raphson
SMIB	Single Machine Infinite Bus
RC	Record creation
ML	Machine Learning
DL	Deep Learning
NN	Neural Network
PMU	Phasor Measurement Unit
PSS	Power System Stabilizer
PDF	Probability Density Function
LogReg	logistic regression
SoftmaxReg	softmax regression
SVM	support vector machines

k-NN	k-nearest neighbors
ReLU	rectified linear unit
IDE	integrated development environment
PSAT	Power System Analysis Toolbox
PST	Power System Toolbox
ANDES	Symbolic Power System Modeling and Numerical Analysis Library
NDA	non-disclosure agreement
API	application program interface
NYISO	New York Independent System Operator
IMF	Intrinsic Mode Function
SMI	Single Mode Index
AMI	All-Mode Index
GMI	Global Mode Index
SSA	small-signal stability analysis
CNN	convolutional neural network
t-CNN	time convolutional neural network
MCDCNN	multi-channel deep convolutional neural network
TSC	time series classification
MLP	multi-layer perceptron
FCN	fully convolutional neural network
HIL	Hardware-in-the-Loop
CNN	Convolutional Neural Network

IoT	Internet of Things
PMU	Phasor Measurement Unit
ML	Machine Learning
SDK	Software Development Kit
GPU	Graphics Processing Unit
ADC	Analog-to-Digital Converter
API	Application Programming Interface
GPU	Graphical Processing Unit
SDK	Software Development Kit
RNN	recurrent neural network
LMU	Legendre Memory Unit
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Units
DTLTI	discrete-time linear time invariant
LTI	linear time invariant
CT	continuous-time
ODE	ordinary differential equation
DT	discrete-time
DE	difference equation
GORU	Gated Orthogonal Recurrent Unit
scoRNN	scaled Cayley orthogonal recurrent neural network
BRC	Bistable Recurrent Cell

nBRC	Neuromodulated Bistable Recurrent Cell
NRU	Non-saturating Recurrent Unit

ACKNOWLEDGMENT

“*Muss es sein? Es muss sein!*”

Beethoven, Quartet No. 16 in F major, Op. 135.

This thesis represents the result of three and a half years of research efforts as part of the ALSETLab. I want to thank Dr. Luigi Vanfretti for allowing me to pursue graduate studies in the United States after recruiting a young and motivated engineer from Bogotá, Colombia as a teaching assistant.

I do not have enough words to thank my co-authors for all their efforts, hard work, and dedication to helping me complete each of the research articles in this document. Giuseppe Laera, Marcelo de Castro Fernandes, Fernando Fachini, Manuel Navarro-Catalán, and Dr. Tetiana Bogodorova are present in most of the papers of this thesis. Without your constant support and collaboration, none of these articles would have passed the first review stage. Shunyao Xu deserves a special mention for his hard work that led to the excellent results in Chapter 7 and 8. Dr. Bhanu Vinzamuri was my mentor for the outstanding results in Chapter 9. Thanks for being an incredible researcher and an even greater human being. To Colleen Smith, thanks for your support and your encouragement to finish this chapter of my life in the best possible way.

Finally, this work is 1000% dedicated to my beloved family. To my lovely and wise grandparents, Meryam and José, and the memory of my grannie “Citica.” To my always-smiling mom Zamira and my funny and smart uncle Yamil. To the loving memory of my dad, “Memo.” To my brothers Daniel, Juan Camilo, and Santiago. To my lovely sisters-in-law Carolina, Karina, and Catalina. To my best friends Nicolás, Jorge, and “Mono” (Juan Carlos). To Simón, Felipe, Elkin, Diana, Fabio, Daniel, Sebastián, Angélica, Claudia, Fernando, “Lala,” Mariana, Irene, Meaghan, Duncan, Māra, Balaji, Ash, Jaime, Iole, and the piccola Aria! I am blessed for being surrounded by you all. “*¿Cómo no os voy a querer?*”

But especially and most importantly, thanks to my one and only, my biggest fan, and my best friend: la mia magica ragazza, my wife, my most heartfelt love, Sofía.

Danke schön. Grazie mille. Gràcies. Merci beaucoup. Thank you. Gracias!

ABSTRACT

Driven by climate change, power system engineers are developing solutions to integrate renewable energies. Such modernization of electrical networks requires novel technologies to increase the efficiency of the electrical generation processes, making them greener and more sustainable. Developing new algorithms and methods is not agnostic to the implosion and success of data-driven strategies in science and engineering. This thesis collects different innovations to facilitate the development of new data-driven algorithms by exploiting physics-based modeling, simulation technologies, and modern computing technologies. The resulting software tool uses models built with the Modelica language and the Python programming language. First, a pipeline for synthetic data generation for electric power transmission systems is described. Deep insight is given into the structure of the different modules and the rationale behind their implementation. Next, the developed tool is used for the implementation of data-driven algorithms. Two case studies corresponding to relevant applications for power systems are presented: small-signal stability assessment and forced oscillation detection. For applications that will require the execution of Machine Learning algorithms at the edge, a low-cost hardware platform is introduced for oscillation detection, which has promising potential for education and research. Finally, to better exploit the embedded physics in electric power transmission systems, a novel recurrent neural network architecture inspired by dynamical systems is presented. Such physics-aware solution promises to play an important role in devising data-driven solutions required for the operation, planning, and control of 21st-century power systems.

CHAPTER 1
INTRODUCTION

Climate change is transforming the electricity generation landscape. The demand to drive away from fossil fuels transforms how electricity is generated and consumed by incorporating greener and sustainable generation. Likewise, consumers get a more active role in electricity markets by turning into prosumers: selling the surplus of electricity they generate at certain hours of the day. This ongoing "mutation" represents a paradigm shift from the hierarchical, vertically-integrated power grid to a distributed smart grid.

Several new technical challenges have arisen in the planning and operation of modern electrical systems [1]. To provide feasible and cost-effective solutions, the engineers of the 21st-century must leverage concepts from optimization [2], controls [3], and data science [4], while benefiting from the continuous advances in computing power and software development. In this fashion, Machine Learning (Machine Learning (ML)) emerges as a promising framework for developing data-driven solutions to assist power system operators.

The Modelica language is one of the most promising means for physics-based simulation tasks for engineers in the 21st-century. Modelica leverages the power of equation-based modeling and offers a natural way of representing dynamical systems while incorporating state-of-the-art numerical solvers. Modelica-based tools are critical in sectors developing cutting-edge technologies, such as the automotive and the aerospatial industries. In the power systems domain, Modelica has been gaining popularity among industry and academic stakeholders for phasor time-domain simulations thanks to the Open-Instance Power System Library (OpenIPSL). OpenIPSL allows the user to benefit from the Modelica language while preserving the structure existing in Power System Simulator for Engineering (PSS®E), the industry standard for such kinds of dynamical studies [5].

The main contribution of this thesis is a pipeline to develop data-driven solutions for modern power systems exploiting the Modelica language. In particular, a pipeline for synthetic data generation based upon massive Modelica simulations is presented. The resulting data is used to train ML-based algorithms for small-signal stability assessment. A byproduct of this thesis is a Python-based tool for automated simulation-based data generation called `ModelicaGridData`, introduced in Chapter 6.

In addition, the results in Chapters 6 to 8 demonstrate the potential of ML techniques for the solution of contemporary problems in power systems such as small-signal stability assessment and forced oscillation detection. Finally, 9 presents a new recurrent neural network (RNN) architecture, inspired by dynamical systems, with a promising potential for time series-driven applications such as monitoring and forecasting.

The thesis summarizes the research outcomes of nine scientific publications first-authored by the student, under the supervision of the thesis advisor and the extensive collaboration of the associated research group. An overview of the thesis is as follows:

- Chapter 2 describes a comparison between different Modelica integrated development environments (IDEs) tailored especially for power system applications. The full materials of this chapter can be found in:
  - S. A. Dorado-Rojas, M. Navarro Catalán, M. de Castro Fernandes, and L. Vanfretti, "Performance Benchmark of Modelica Time-domain Power System Automated Simulations using Python," presented at the Proceedings of the American Modelica Conference 2020 [6];
- Chapter 3 proposes a novel data structure to handle power flow variables required to change the initial conditions of dynamic simulations. The associated publication of this chapter is:
  - S. A. Dorado-Rojas, G. Laera, M. de Castro Fernandes, T. Bogodorova, and L. Vanfretti, "Power Flow Record Structures to Initialize OpenIPSL Phasor Time-Domain Simulations with Python," presented at the 14th International Modelica Conference 2021 [7];
- Chapter 4 introduces an ad-hoc methodology for contingency generation to produce generic but realistic scenarios for data generation. Likewise, this chapter shows a case study related to the development of "classical" ML algorithms. The reference publication for this section of the document is:
  - S. A. Dorado-Rojas, M. de Castro Fernandes, and L. Vanfretti, "Synthetic Training Data Generation for ML-based Small-Signal Stability Assessment," presented at the 2020 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm) [8];

- Chapter 5 describes the software tool used to automate the power flow computation, contingency generation, and simulation dispatch using Modelica and Python. The key publication for this chapter is:
  - S. A. Dorado-Rojas, F. Fachini, T. Bogodorova, G. Laera, M. de Castro Fernandes, and L. Vanfretti, “ModelicaGridData: Massive Power System Simulation Data Generation and Labeling Tool using Modelica and Python,” submitted to the journal SoftwareX in June 2022 [9];
- Chapter 6 provides insight into a potential application of the synthetic data streams, namely, in time series-based small-signal stability assessment for power systems using Deep Learning (DL) methods. The main contributions of this chapter were presented in:
  - S. A. Dorado-Rojas, T. Bogodorova, and L. Vanfretti, “Time Series-Based Small-Signal Stability Assessment using Deep Learning,” presented at the 2021 North American Power Symposium [10];
- Chapter 7 describes the development of a low-cost hardware platform for evaluating and validating ML prototypes used to detect forced oscillations in power systems. These results were included in the article:
  - S. A. Dorado-Rojas, S. Xu, L. Vanfretti, G. Olvera, M. I. I. Ayachi, and S. Ahmed, “Low-Cost Hardware Platform for Testing ML-Based Edge Power Grid Oscillation Detectors,” presented at the 2022 10th Workshop on Modelling and Simulation of Cyber-Physical Energy Systems (MSCPES) [11];
- Chapter 8 elaborates on the algorithm selection for the forced oscillation detection techniques evaluated with the platform in Chapter 7. The research of this chapter is included in the following publication:
  - S. A. Dorado-Rojas, S. Xu, L. Vanfretti, M. I. I. Ayachi, and S. Ahmed, “ML-Based Edge Application for Detection of Forced Oscillations in Power Grids,” presented at the 2022 IEEE Power & Energy Society General Meeting [12];
- Chapter 9 switches gears and veers toward a more theoretical contribution developing a dynamical systems-inspired RNN architecture. The ideas in this chapter are part of the article:

– S. A. Dorado-Rojas, B. Vinzamuri, and L. Vanfretti, "Orthogonal Laguerre Recurrent Neural Networks," presented at the 34th Conference on Neural Information Processing Systems (NeurIPS) [13];

- Finally, Chapter 10 concludes the work.

Each of the chapters in this document is an outcome of the research efforts made for this thesis. All corresponding articles have been peer-reviewed for presentation at a conference or publication in a journal. Besides, the results in Chapter 9 have been patented as part of a collaboration between Rensselaer Polytechnic Institute and IBM. A guide for the reader navigating the document is shown in Fig. 1.1.

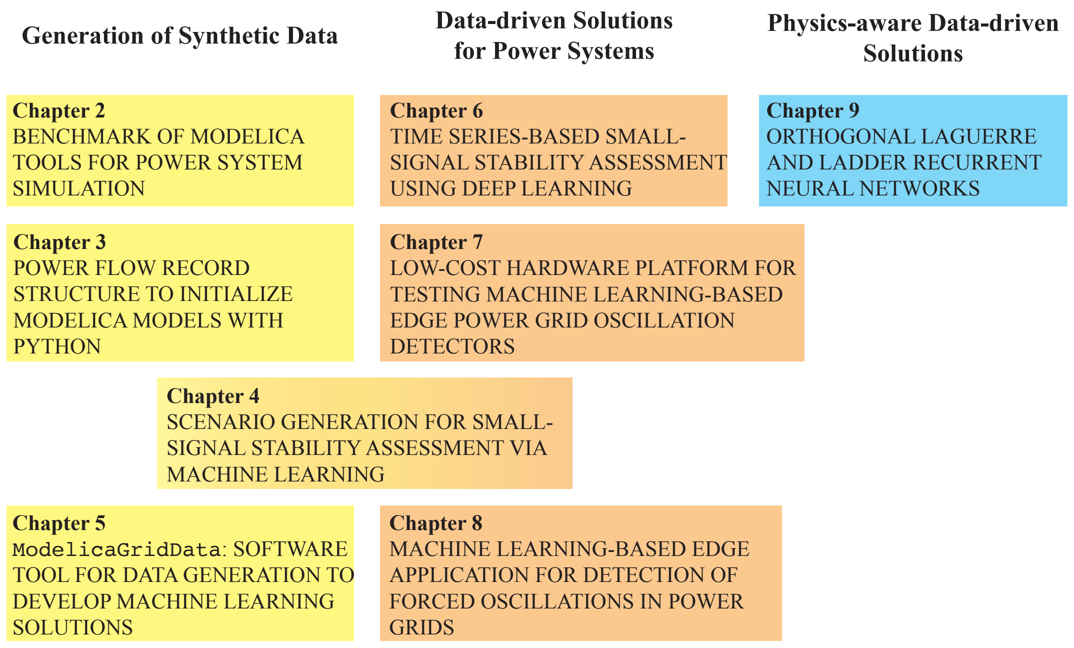

Generation of Synthetic Data
Data-driven Solutions for Power Systems
Physics-aware Data-driven Solutions
Chapter 2 BENCHMARK OF MODELICA TOOLS FOR POWER SYSTEM SIMULATION
Chapter 6 TIME SERIES-BASED SMALL-SIGNAL STABILITY ASSESSMENT USING DEEP LEARNING
Chapter 9 ORTHOGONAL LAGUERRE AND LADDER RECURRENT NEURAL NETWORKS
Chapter 3 POWER FLOW RECORD STRUCTURE TO INITIALIZE MODELICA MODELS WITH PYTHON
Chapter 7 LOW-COST HARDWARE PLATFORM FOR TESTING MACHINE LEARNING-BASED EDGE POWER GRID OSCILLATION DETECTORS
Chapter 4 SCENARIO GENERATION FOR SMALL-SIGNAL STABILITY ASSESSMENT VIA MACHINE LEARNING
Chapter 5 ModelicaGridData: SOFTWARE TOOL FOR DATA GENERATION TO DEVELOP MACHINE LEARNING SOLUTIONS
Chapter 8 MACHINE LEARNING-BASED EDGE APPLICATION FOR DETECTION OF FORCED OSCILLATIONS IN POWER GRIDS

Figure 1.1: Structure of the thesis document.

CHAPTER 2
BENCHMARK OF MODELICA TOOLS FOR POWER SYSTEM SIMULATION

2.1 Introduction

Modeling and simulation of power systems have been a common practice in the energy industry since the 1960s. The complexity of a power system is steadily increasing to accommodate modern technologies into the existing grid. A more complex system leads to more elaborated models. High-complexity models are directly correlated with computationally expensive tasks [14]. In this context, the Modelica language represents a modern equation-based, multi-domain solution modeling and simulation alternative. Numerous initiatives such as OpenIPSL have been taken to bring to the power system domain the benefits of the Modelica language [15].

On the other hand, the academic, scientific, and industrial communities have come to acknowledge the intrinsic benefits of the Modelica language. An outcome of this trend is that the user base has increased significantly during the last few years. This has led to the development of many libraries with users coming from a very wide domain spectrum. Nowadays, Modelica stakeholders include students, consulting firms, large laboratories, and industrial institutions.

Cost-free tools such as OpenModelica (OM) are fundamental for learning the language at little to no cost and to set a reference for the Modelica language [16]. Commercial tools such as Dymola[1], SystemModeler[2], Modelon Impact[3] or SimulationX[4] provide advanced functionalities that satisfy particular requirements from the industry. However, there is no clear guidance for a user on how to select a particular tool and numerical solver based on its simulation performance exclusively. Providing this guidance is the goal of this chapter.

We aim to compare the time-domain simulation performance of the solvers from both Dymola and OpenModelica (OM) when subjected to different solver settings (see [17] for a

Portions of this chapter appear in S. A. Dorado-Rojas, M. Navarro Catalán, M. de Castro Fernandes, and L. Vanfretti, "Performance Benchmark of Modelica Time-domain Power System Automated Simulations using Python," presented at the American Modelica Conference 2020 [6].

[1]Dymola: https://www.3ds.com/products-services/catia/products/dymola/.

[2]SystemModeler: https://www.wolfram.com/system-modeler/graphical-system-modeling/.

[3]Modelon Impact: https://www.modelon.com/modelon-impact/.

[4]SimulationX: https://www.esi-group.com/products/system-simulation.

detailed analysis of the potential of OM to solve large-scale models). Since these tools do not have the same features and solvers, we have chosen some of the ones they have in common for benchmarking purposes.

The contributions in this chapter are relevant to any user of the Modelica language, especially for power system simulation practitioners. The tool performance analysis is based on the simulation of a commonly used power system model (IEEE 14 bus system), that serves as a representation of a dynamic nonlinear power system. We consider three simulation scenarios: an initialization, a line-opening (one discrete event) and two bus faults (two discrete events). We study the difference in performance within the tools and help users make an educated choice about the tools to use. The main contributions of this chapter are the following:

- quantitative evaluation of Dymola and OM simulation performance for time-domain simulation of complex dynamic systems (power systems);
- benchmark of different solvers in a dynamic simulation with discrete events;
- implementation of simple Python routines to automate Dymola and OM time-domain simulations.

This chapter is broken down in the following sections: Section 2.2 describes the test system and the Modelica library employed to construct it. The experiment setup regarding hardware characteristics and software setup is described in Section 2.3. In Sections 2.4 and 2.5, we discuss the experiment results of each of the tools with respect to each solver and the corresponding performance metrics. Finally, Section 2.6 concludes the chapter.

2.2 Modelica Power System Model

The bus system[5] represents a part of the Midwestern USA American Electric Power System as of February of 1962. The single-line diagram of the system can be seen in Figure 2.1a. This model was chosen because it is a widely used testing system for an initial assessment in power system dynamical studies since it has a significant number of variables and states (420 and 49, respectively) which makes it a common factor in such simulation-based studies [14]. For this reason, its dynamic simulation poses a challenge to the tools and the CPU.

[5]Different models of the IEEE 14 bus system can be found at https://icseg.iti.illinois.edu/ieee-14-bus-system/.

The test power system model (Figure 2.1b) is built using the components from the open-source OpenIPSL library, a Modelica-based power system component library currently developed and maintained by ALSETLab at Rensselaer Polytechnic Institute. The library includes all the components to build a large power system model and perform dynamic analysis in time- and phasor-domain. The version of the library used is release 1.5.0[6].

2.3 Experiment Specifications

To assure that the results are reproducible, this section details the conditions under which the experiments were performed regarding hardware setup and software characteristics.

2.3.1 Hardware and Software Setup

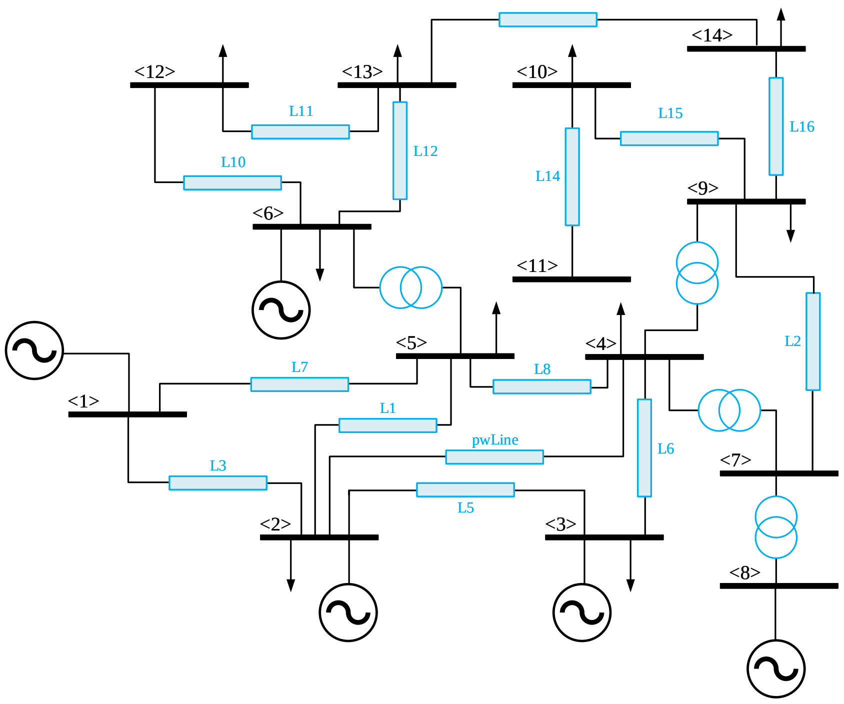
The characteristics of the computer used for the simulations are shown in Table 2.1.

Table 2.1: Hardware characteristics and software specifications of the computer used to run the performance assessment experiments.

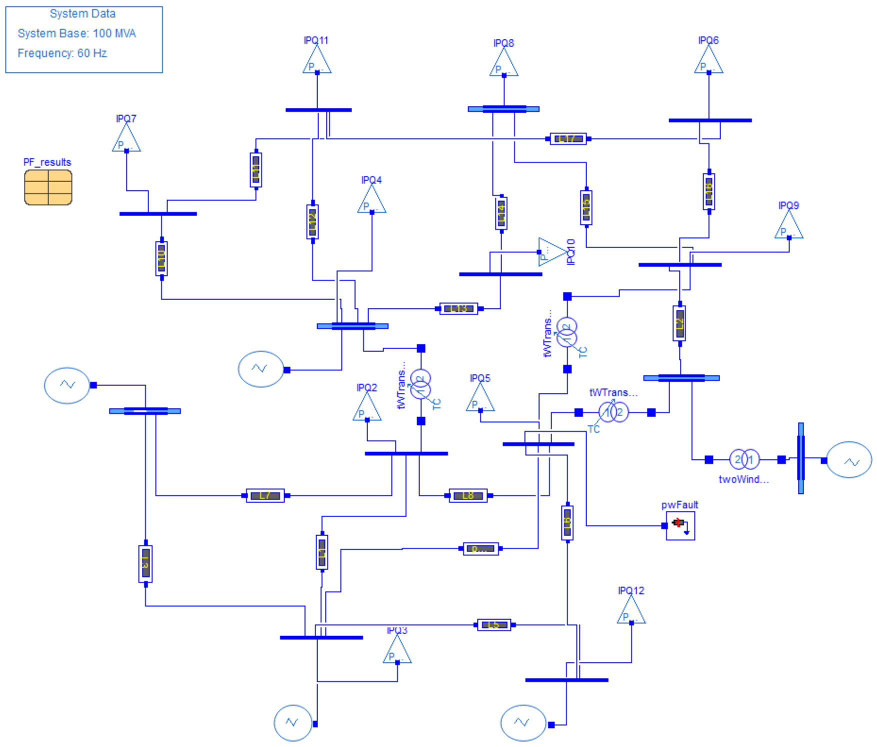
Operating System	Ubuntu Server 18.04 LTS
RAM	128 GB
Processor	Intel(R) Xeon(R) CPU E-1650 v4 12 Cores @ 3.60 GHz 15 MB Cache
Storage	1 TB SSD
Graphics Cards	4 x NVIDIA GTX 1080 Ti (CUDA Capable) 11 GB GDDR5X (each)
Dymola Distribution	Dymola 2020x
OM Distribution	1.14.0
Python Release	3.6.8
Dymola Compiler	MinGW CC
OM Compiler	MinGW CC

To assess solver performance correctly, numerical integration must run in only one processor. While this is a default option in Dymola, we need to specify this option explicitly in OM before starting any simulation since it defaults to multi-core execution. This is done thanks to the flag `setCommandLineOptions("-n=1")`.

[6]The version of OpenIPSL used for these experiments is made available together with the main code of the program on GitHub: https://github.com/ALSETLab/Time-Domain-Simulation-Performance-Benchmark. For the latest release of OpenIPSL, see: https://github.com/OpenIPSL/OpenIPSL.



(a) IEEE 14 bus model.



(b) Implementation of the IEEE 14 bus model in Dymola using OpenIPSL.

Figure 2.1: Single line diagram and Modelica implementation of the IEEE 14 bus system.

2.3.2 Simulation Scenarios

To properly measure solver performance in diverse dynamic conditions, we will consider the following three scenarios of the IEEE 14 bus model: system initialization, time-domain simulation with one line opening, and system response with two faults.

IEEE 14 System Initialization

This scenario, labeled as S_1, corresponds to a simulation with no disturbing events. The power flow condition of the model is modified so that the numerical solver is exposed to difficulties during initialization. The provided initial conditions are such that the dynamic system is not initially at an equilibrium point, thus forcing the numerical solver to look for an acceptable steady-state condition at the beginning of the integration process (what is known as *initialization*). This increases the computational task and challenges the solver since the integration does not start with all state derivatives at zero.

Line Opening

Besides the aforementioned "bad" initialization condition, we force a line opening to disturb the system from steady-state and excite nonlinear dynamics (experiment S_2). This type of scenario is used to study system-wide stability when two sub-areas are disconnected from each other. The line opening corresponds to the connection between buses 2 and 4 (`B2` and `B4`). The line will open from both ends at time $t = 60$ s and will re-close at $t = 61.5$ s.

Bus Faults

In this case (S_3), the system will face two three-phase to ground faults at different times. This configuration is used to test the resiliency and stability of the system. By having two faults, the numerical complexity of the simulation increases, creating a more adverse scenario for the solvers to come up with a solution. Fault 1 occurs at bus 4 (`B4`) starting at $t = 20$ s and being removed at $t = 21.2$ s. Fault 2 takes place at bus 14 (`B14`) at $t = 80$ s, being cleared at $t = 81.2$ s. The parameters of the two faults are $R = 0$ pu and $X = 1 \times 10^{-5}$ pu.

2.3.3 Solver Selection

Performance of a time-domain simulation depends not only on the dynamic condition to be analyzed but also on solver selection. In this regard, OM and Dymola contain a wide variety of different integration methods and three of them are going to be used and thus briefly described in this study. The Differential Algebraic System Solver (`dassl`) is an implicit, high-order, variable-step solver with time-step control. This solver is set as default solver in both OM and Dymola. The `Euler` method is another solver available in both software packages and it is an explicit (i.e., forward Euler), first-order, fixed time-step solver. Finally, the last solver used in this study is the `runge kutta`. Dymola allows the user to chose between second, third and fourth order Runge-Kutta methods but in this work, only the fourth order is used because it is also available in OM. This solver is an explicit, fourth-order, fixed time-step solver. We will benchmark the performance of the tools with each of the mentioned solvers for the different scenarios of the test power system.

2.3.4 Time-step Selection

Since `dassl` is a variable-step solver with step-size control, there is no need to select a specific time-step for the simulation. The selection of an adequate number of intervals is necessary to plot and analyze the results. For both tools, 5,000 was found to be a reasonable number of simulation intervals. This means that after the simulation is complete, the program will interpolate the results to give an output with 5,000 points. Moreover, to use the capabilities of a Differential Algebraic Equation (DAE) solver to their full extent, we enable the newly incorporated `DAEmode` in Dymola by enabling the flag `Advanced.Define.DAEsolver = true` [18]. In OM, to set similar settings we use the command `setCommandLineOptions("daeMode=true")`.

On the other hand, it is important to select an adequate step size T_s for fixed-step solvers in order to guarantee that the algorithm is operating in its region of numerical stability. To get an upper bound for T_s, we performed a linear analysis of the system in Dymola employing the library `Modelica_LinearSystems2`. After determining the time constant of the fastest mode ($\tau \approx 1$ ms), we found that $T_s = 0.5$ ms was a reasonable value to capture the effects of the fastest mode, guaranteeing numerical convergence for both solvers, `Euler` and `Runge Kutta`. The selected time-step size implies that 240,000 simulation intervals are going to be needed for a simulation time of 120 s.

2.3.5 Benchmark Metrics

In order to understand and accurately compare the two tools, the experiments focus on two simulation features to compare:

- simulation time (ST) corresponds to the time it takes for a program to complete all of the routines for each scenario comprising model translation, compilation and execution. The discussion of the results of ST are found in Section 2.4.1, with special remark on execution time (ET).
- *CPU Utilization* is the percentage of CPU that is being used at any time during the execution. Results for CPU utilization can be found in Section 2.4.2.

2.3.6 Code Structure

The complete code to perform the experiments and analyze the resulting data can be found in GitHub[7]. The execution of the simulations is automated through Python using the Python-Dymola Interface (PDI) and the OpenModelica-Python Interface (OMPython) [19]. The details of the Dymola routine can be seen in the file `dymola_simulation.py`. Likewise, the OM commands are included in the file `om_simulation.py`.

The routine in the script `measurement_performance.py` measures the computing performance. It registers each of the performance metrics every 0.2 s while the code is running in a different parallel process. The main program is contained in the file `01_modelica_tool_performance_benchmark.py`.

2.4 Performance Results

Before presenting the performance results, we validate the simulation outputs of the three scenarios for Dymola and OM for all solvers. We employed the Normalized Absolute Error (NAE) and the Mean Squared Error (MSE) defined in Equation (2.1) to quantify the numerical difference between the outcomes of each tool:

$$\begin{aligned} \text{NAE} &= \frac{|x_i - y_i|}{n} \\ \text{MSE} &= \sum_{i=1}^{n} \frac{(x_i - y_i)^2}{n}. \end{aligned} \tag{2.1}$$

[7]https://github.com/ALSETLab/Time-Domain-Simulation-Performance-Benchmark.

NAE shows how different the Dymola and OM results are throughout the simulation. MSE outputs a quantitative validation of the results of both tools [20]. Full details can be seen in Table 2.5.

The numerical behavior of the simulation during initialization (`runge kutta` solver) can be observed in Figure 2.2 for the voltage magnitude signal at `B2` and `B4`. An initial transient behavior can be seen at the beginning of the integration time. This is not desired in a dynamic simulation since numerical convergence to a steady-state solution is not guaranteed given the fact that the solver starts from a guessing point with non-zero derivatives.

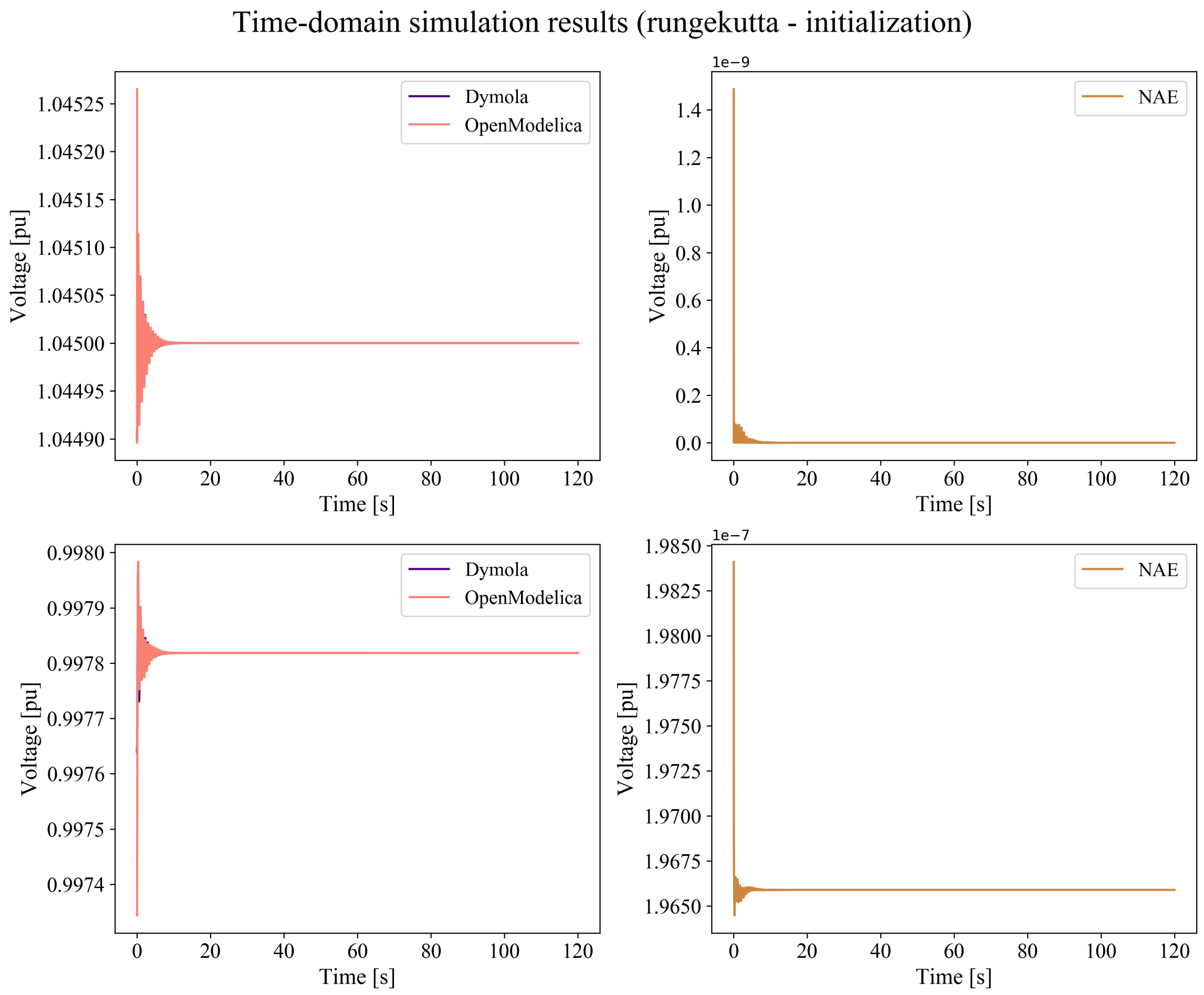


Figure 2.2: Comparison between Dymola and OM results for the initialization scenario using the `runge kutta` solver.

The non-steady state behavior at the on-set of the simulation is due to the fact the initial guess used in the model (the power flow condition) is not close enough to an equilibrium for the initialization routine to solve for a more precise set of initial values. A more complex initialization problem will better benchmark the capabilities of the tools. Despite this, Dymola and OM produce almost the same results, with an NAE in the order of 10^{-7}.

Likewise, for the `runge kutta` solver, Figures 2.3a and 2.3b show the simulation results for the line opening (voltage magnitude at `B2` and `B4`) and the double bus fault (voltage at affected buses `B4` and `B14`) scenarios, respectively. Both figures reveal how there is a minimal error between the results of both tools. Based on these results, it is concluded that fixed-step solvers can be applied to reduce discrepancies between different Modelica tools.

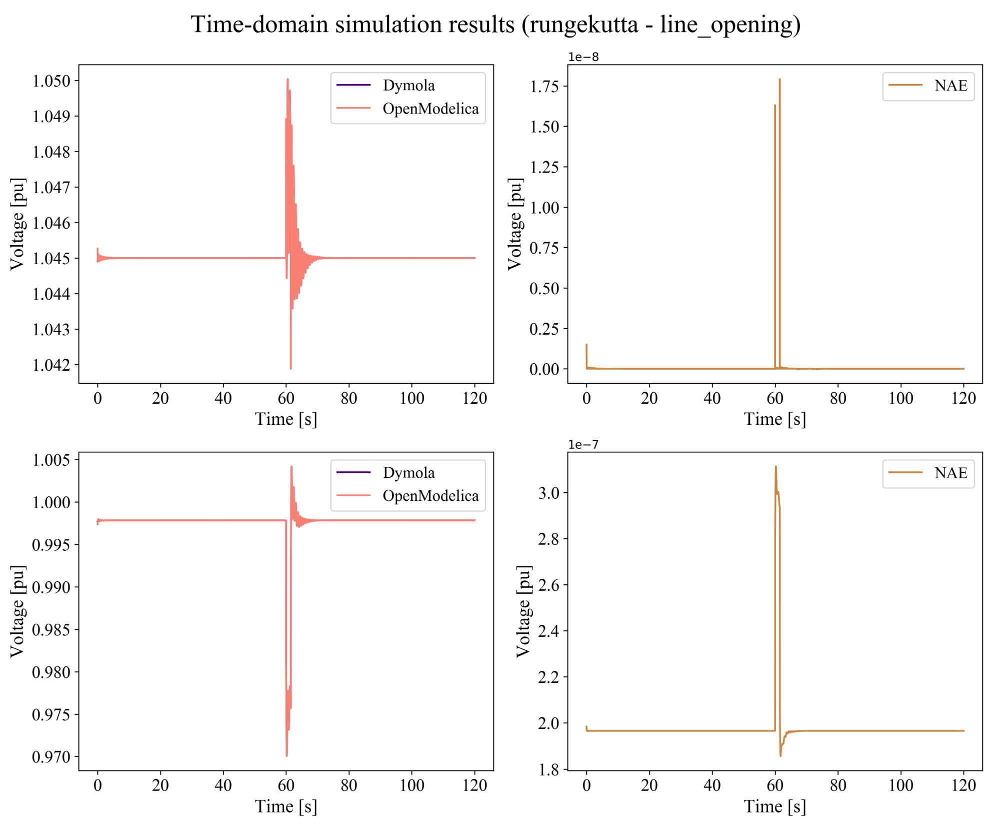
2.4.1 Simulation Time

The information regarding simulation time is presented for all scenarios and solvers in Table 2.4. We must underline that simulation time includes compilation, translation, and the actual numerical integration (execution time). A clear conclusion from this information is that the variable-step solver is the most convenient for an initial analysis of the conditions of the system with an important amount of detail. Nevertheless, considering the information about MSE, a fixed-step solver shows advantages to reduce the numerical discrepancy between tools running the same model. The cost is an increase in simulation time.

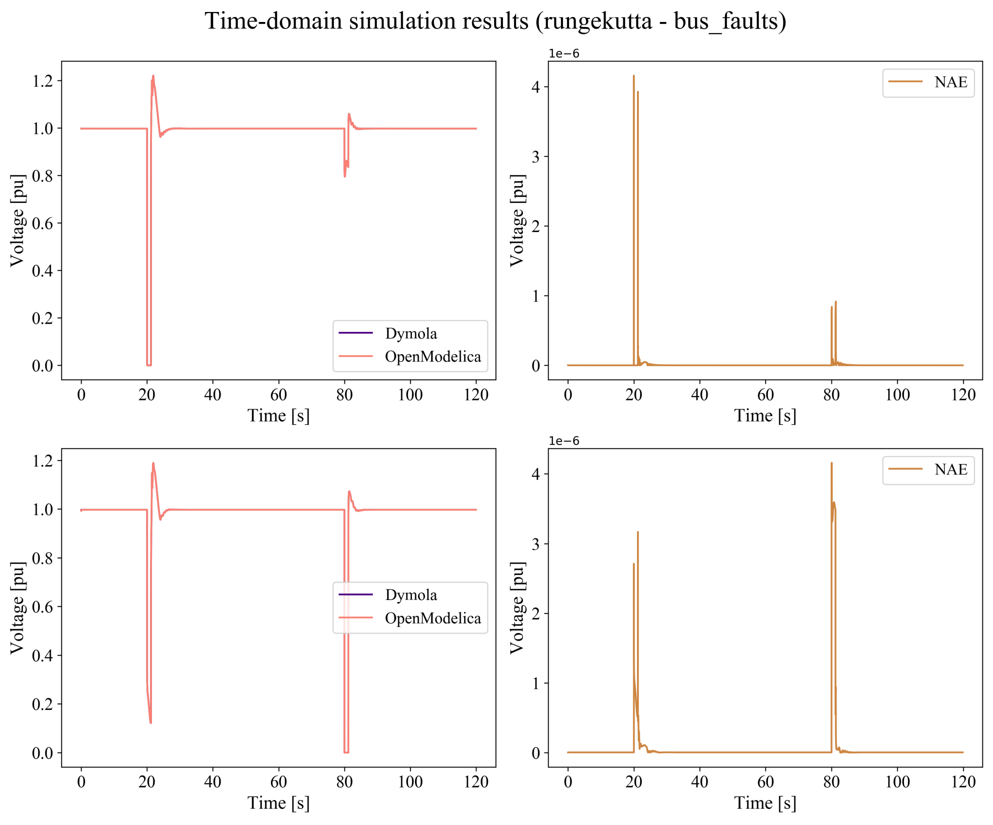
2.4.2 CPU Utilization

Since each instance of Dymola/OM was constrained to run only on one core, we expect exactly one processor to be responsible for numerical integration while a simulation is being carried out. The CPU usage of the assigned execution core is 100% due to the heavy numerical task of the simulation.

An interesting outcome of our experiments is that several CPUs are involved in the execution process but just one is performing the simulation tasks at a given time. We can detail this behavior in Figure 2.4a for a Dymola simulation using the `runge kutta` method of the bus fault scenario. Simulation starts in Core 1 where the CPU usage is at a 100% at the beginning of the running time. Afterward, it is delegated to Core 5. Finally, Core 10 completes the execution of the program. This behavior is due to a task scheduling routine in the processor level that dispatches to different cores the compilation, translation, and integration sub tasks. Similar behavior happens with another solver and OM (Figure 2.4b) in which the simulation started in Core 2, then was briefly assigned to Core 5 and was finished in Core 6. All the plots can be detailed in the GitHub repository inside the Jupyter Notebook called `03_DataPostprocessing_CPU_Usage.ipynb`.

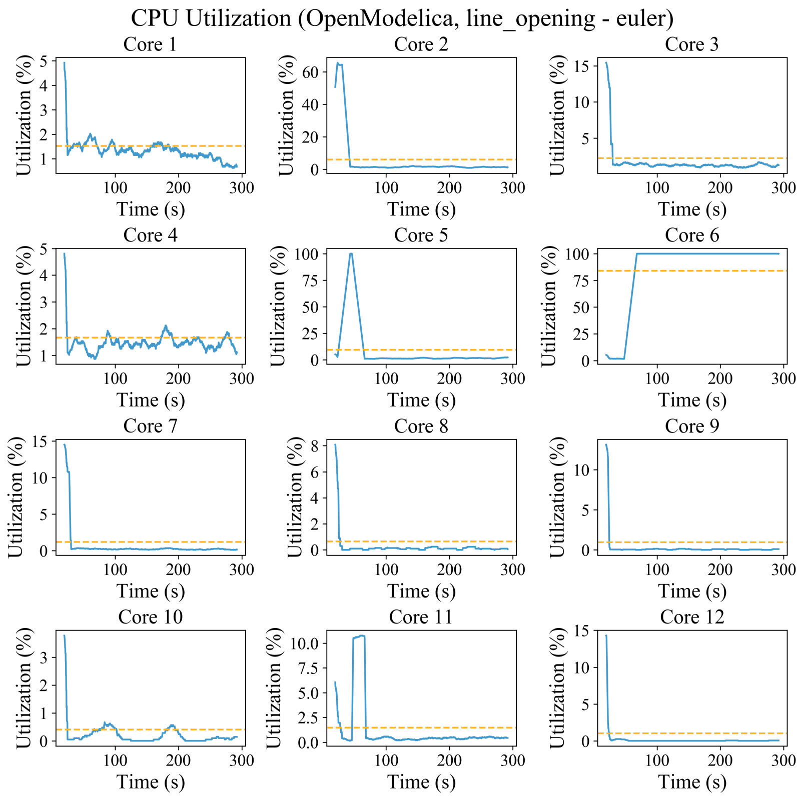


(a) Line opening scenario (S_2).

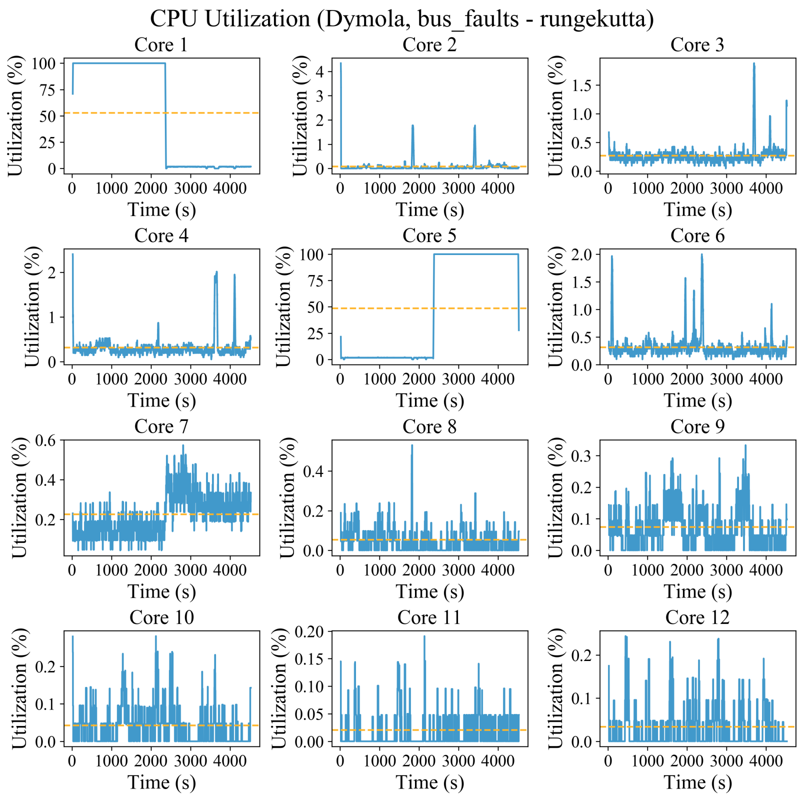


(b) Bus fault (S_3) scenario.

Figure 2.3: Comparison between Dymola and OM results for different simulation scenarios using the `runge kutta` solver.



(a) OM



(b) Dymola.

Figure 2.4: CPU utilization for Dymola and OM during line opening and bus fault scenarios with the `runge kutta` solver.

2.5 Performance Evaluation Metrics

A score was proposed to quantify the performance differences between the tools and the solvers. The score is obtained from the data generated for all simulations and solvers. This single metric makes it simpler to directly compare the performance of Dymola versus OM. From Table 2 the ET for each scenario and solver were employed. These metrics were obtained directly from the program logs and measured in Python. Notice that the time registered using OMPython is slightly larger than the reported by the simulation log due to the communication delay between Python and OM. The translation and compilation time were not taken into account since this information was only available in the Dymola developer version, not in the release version at the time of writing this document.

The normalized minimum execution time (NMT) score of each scenario per solver is computed as

$$\mathrm{NMT}^{[\mathrm{solver}]} = \frac{\min(\mathrm{ET_D}, \mathrm{ET_{OM}})}{\mathrm{ET_{observed}}} \tag{2.2}$$

where $\mathrm{ET_{observed}}$ is the ET for a particular solver in Dymola or OM, and $\min(\mathrm{ET_D}, \mathrm{ET_{OM}})$ is the minimum execution time between both tools for a specific solver. Clearly, $\mathrm{NMT}^{[\mathrm{solver}]}$ lies between 0 and 1. The higher the NMT is, the faster the simulation will run for a particular selected solver. At a first glance, this metric might be counter-intuitive since a better solver/tool combination would reduce execution time. However, we propose an increasing score metric due to the fact that users are more familiar to higher scores for better performance. Therefore, the larger the NMT is, the faster a particular solver will run.

The NMT metric results are presented in Table 2.2. The performance of Dymola is remarkably better using `dassl`. Nevertheless, OM shows a smaller execution time than Dymola for fixed-step solvers (as can be seen from Table 2.4, the NMT scores and 2.3). This conclusion can be further detailed in Table 2.3 where a direct comparison between the execution time for the tools with the different solvers for each scenario is presented.

The NMT scores highlight that the performance of Dymola in terms of execution time is remarkably better for variable-step solvers. The relative advantage of selecting one tool with respect to the other can be computed from the NMT directly. For instance, Dymola runs 47.3x faster than OM for the first scenario using `dassl` which can be computed by a direct comparison of the ET listed in Table 2.3. The NMT score of OM for S_1 is 0.0211

Table 2.2: NMT scores.

	Dymola			OM		
	$\mathrm{NMT}^{[S_1]}$	$\mathrm{NMT}^{[S_2]}$	$\mathrm{NMT}^{[S_3]}$	$\mathrm{NMT}^{[S_1]}$	$\mathrm{NMT}^{[S_2]}$	$\mathrm{NMT}^{[S_3]}$
`dassl`	1	1	1	0.0211	0.0254	0.0880
`euler`	0.148	0.168	0.208	1	1	1
`runge kutta`	0.177	0.296	0.293	1	1	1

which is $1/0.0211 = 47.3$ times smaller than the corresponding Dymola metric reflecting the relative difference in execution time.

For the variable-step solver, the discrepancy between the tools can be attributed to the performance of the `dassl` solver in all simulations thanks to the aforementioned improvements for `DAEMode` inside Dymola [18].

The execution time of OM is faster than the one of Dymola for all scenarios when a fixed-step solver is used. The NMT scores show a relative advantage between 3.4 and 6.8 times favoring OM. We have contacted Dassault Systèmes about the performance of the simulations of the IEEE 14 Bus System using `runge kutta` methods (including `euler`) as integrator and GCC for compilation. Dassault reports that bug fixes have been made in the GCC runtime libraries, leading to CPU times are about 3 – 4 times faster, on par with the run times given when compiling with Visual Studio under Windows 10. Dassault informs that the updated libraries were included in Dymola 2021.

We should point out that the scope on the potential optimization features has been limited to the use of the flag `Evaluate = true` in Dymola and `-d=evaluateAllParameters` in OM, which is standard practice when attempting to improve simulation performance.

The detailed step-by-step computations of the scores can be found in GitHub in the Notebook `05_BenchmarkMetrics.ipynb`.

2.6 Conclusions

The chapter presented a concrete analysis of the time-domain simulation performance of Modelica-based tools for different solvers in the context of large-scale nonlinear dynamic systems. The presented results can help a user to choose a tool depending on the final application, and lead to improvements in Modelica tools. The methodology of this benchmark can be extended to virtually any platform or Modelica tool.

We benchmarked the time-domain simulation performance of two popular Modelica tools, Dymola and OM, for a dynamic power system simulation using the IEEE 14 bus system. We considered several scenarios that challenge numerical solvers differently. Thanks to Python scripting, we were able to change automatically the simulation settings while directly measuring the performance of the computer instead of relying on simulation logs. Python functions also made it quicker to analyze straightforwardly the big set of data regarding simulation results and computer performance.

For the proposed heuristic score, we found out that OM performs better than Dymola in terms of execution time for fixed-step solvers while Dymola shows faster results when using a variable-step solver (see Table 2.3). Despite this, we must warn the reader that this conclusion is based upon only a particular system. Further research has to be done to include more test systems. However, the given GitHub code can be used as a driver to conduct testing on other power system models.

In Tables 2.4, 2.5, and 2.3, `runge kutta` is abbreviated as `rk`.

Table 2.3: Performance comparison between Dymola and OM.

		Execution time (ET)							
		OM	**Dymola**	**Benchmark Result**			**OM**	**Dymola**	**Benchmark Result**
S_1	`dassl`	7.869 s	0.1664 s	D > OM (47.3x)	S_3	`dassl`	163.48 s	14.40 s	D > OM (11.3x)
	`euler`	277.54 s	4420.01 s	OM > D (6.8x)		`euler`	378.60 s	1820.01 s	OM > D (4.8x)
	`rk`	783.01 s	1880.01 s	OM > D (5.6x)		`rk`	1344.68 s	4590.01 s	OM > D (3.4x)
S_2	`dassl`	13.40 s	0.3408 s	D > OM (39.3x)					
	`euler`	310.10 s	1850.01 s	OM > D (6.0x)					
	`rk`	1086.39 s	4410.01 s	OM > D (4.1x)					

Table 2.4: Execution time for Dymola and OM for each simulation scenario using different solvers.

		Simulation Time with OpenModelica (OM)				
		Translation	Compilation	Execution	Total Time (OM log)	OMPython
S_1	`dassl`	2.3204 s	6.6270 s	7.8690 s	16.8164 s	19.3451 s
	`euler`	2.5432 s	6.5845 s	277.5495 s	286.6772 s	289.2878 s
	`rk`	2.3495 s	6.6213 s	783.0159 s	791.9867 s	794.6805 s
S_2	`dassl`	2.6079 s	6.6411 s	13.4004 s	22.6494 s	25.2542 s
	`euler`	2.4023 s	6.6437 s	310.1061 s	319.1521 s	321.7222 s
	`rk`	2.3489 s	6.6591 s	1086.3958 s	1095.4040 s	1098.0253 s
S_3	`dassl`	2.1952 s	6.7301 s	163.4884 s	172.4137 s	175.2962 s
	`euler`	2.3248 s	6.7801 s	378.6069 s	387.7118 s	390.3140 s
	`rk`	2.3960 s	6.7332 s	1344.6808 s	1353.8100 s	1356.2994 s

		Simulation Time with Dymola						
		Translation + Compilation	Execution	Measured Python		Translation + Compilation	Execution	Measured Python
S_1	`dassl`	20.186 s	0.1664 s	20.3524 s	S_3 `dassl`	20.2161 s	14.4098 s	34.6260 s
	`euler`	24.7791 s	1880.0109 s	1904.7900 s	`euler`	16.4581 s	1820.0119 s	1836.47 s
	`rk`	21.2389 s	4420.0125 s	4441.2514 s	`rk`	17.9956 s	4590.0129 s	4608.0085 s
S_2	`dassl`	20.2363 s	0.34082 s	20.5772 s				
	`euler`	19.6561 s	1850.0129 s	1869.6690 s				
	`rk`	24.6567 s	4410.0119 s	4434.6686 s				

Table 2.5: MSE between voltage magnitude signals at different buses for each simulation scenario.

		Mean Squared Error (MSE)				
		B2	B4		B1	B4
S_1	`dassl`	3.0011×10^{-11}	4.6482×10^{-11}	S_3 `dassl`	0.0067	0.0002
	`euler`	1.2894×10^{-11}	3.7950×10^{-11}	`euler`	0.0025	0.0002
	`rk`	1.2853×10^{-11}	3.7828×10^{-11}	rk	0.0018	0.0002
S_2	`dassl`	1.1728×10^{-8}	1.1267×10^{-7}			
	`euler`	2.3598×10^{-10}	3.2470×10^{-9}			
	`rk`	2.3579×10^{-10}	3.2473×10^{-9}			

CHAPTER 3
POWER FLOW RECORD STRUCTURE TO INITIALIZE MODELICA MODELS WITH PYTHON

3.1 Introduction

The Open-Instance Power System Library (OpenIPSL) is an open-source library of power system component models written entirely in Modelica [15]. Beyond the inherent advantages of the Modelica language, OpenIPSL components are constantly cross-validated against commercial packages such as PSS®E, producing practically the same results [5] and exhibiting the same or even better simulation performance (see [18] and [6]).

Successful use cases of the library come from a broad range of applications such as multi-domain simulation [21], damping [22] and parameter estimation [23] in power systems, dynamic stability assessment [24], co-simulation for energy analysis [25], stability analysis of hydro-power grids [26], wind turbine control [27], cyber-attack evaluation [28], power system stability enhancement [29], extremum seeking control [30], and data generation for machine learning applications [8].

Despite the library's usefulness, the main caveat is the absence of a systematic approach to link phasor time-domain simulations with static computations like power flows. Power flow computations are ubiquitous in any power system analysis. A power flow problem involves determining the system's voltage profiles and electrical power transfer across a network given the generator power injections and load consumption. Mathematically, it is a nonlinear vector algebraic equation commonly solved using an iterative method such as a Newton-Raphson (NR) algorithm. From the dynamical perspective, a power flow result represents an operating condition for which may contain a *potential* equilibrium for the underlying dynamical system. So, the power flow result represents the set of initial guesses to initialize a dynamic model and analyze an electrical grid's behavior subjected to a dynamical event. Observe that because

Portions of this chapter appear in S. A. Dorado-Rojas, G. Laera, M. de Castro Fernandes, T. Bogodorova, and L. Vanfretti, "Power Flow Record Structures to Initialize OpenIPSL Phasor Time-Domain Simulations with Python," presented at the 14th International Modelica Conference 2021 [7].

the simplified algebraic representation of the power grid in the power flow problem, many of its solutions can result in operating conditions where an equilibrium may not exist when the system's dynamical model is considered.

OpenIPSL models contain a myriad of nonlinearities employed to represent dynamical behaviors more accurately. So, varying the initial condition of a dynamical simulation represents a critical step towards system assessment. So far, users have proposed ad-hoc solutions to generate power flow results (e.g., using Matpower[1] or PSS®E[2] as in [31]). However, despite valuable, these efforts do not completely fill the gap to easily provide power flow solutions to OpenIPSL models. The former approach replaces the power flow values in the `*.mo` file of the model directly, which is inconvenient from the user's point of view. The latter depends on proprietary software which might not be available to the base users of OpenIPSL. The OpenIPSL community, and users of other Modelica-based power system libraries (see [32]), will more than welcome a systematic power flow approach based on open-source tools to integrate into their models quickly. Addressing this issue is the primary purpose of this chapter. The main contribution is to bridge the gap between phasor time-domain simulation and static computations for OpenIPSL utilizing an open-source-based pipeline.

We propose a Modelica records structure to handle all power flow variables. Such nested records data structure enables a user to replace a power flow condition, typically composed of several algebraic variables, with a single click or one line of code. These records are created automatically from the model's `*.mo` file and populated using GridCal.

GridCal[3] is an open-source Python library for power system computations such as power flows. A remarkable characteristic of GridCal is its built-in PSS®E parser. Consequently, users can parse `*.raw` files containing a grid static model's information to a GridCal internal grid representation. In our examples, we will use PSS®E files to construct GridCal models. This enables us to benchmark the GridCal power flow results against PSS®E outputs. By doing so, we bring confidence to Modelica tools in terms of the quality of results, showing that Modelica-based power system models can be initialized and

[1]See https://github.com/dgusain1/InitialiseModelica

[2]See https://github.com/ALSETLab/Raw2Record

[3]GridCal is able to import and parse model description and parameter files from proprietary software such as PSS®E and DigSilent, and also widespread open-source libraries for power system analysis like Matpower. In contrast to many proprietary electrical grid software tools, GridCal runs on Windows, Linux, and macOS natively. It can be downloaded from https://github.com/SanPen/GridCal.

result in the same initial condition and provide the same simulation results as proprietary tools.

So far, we summarize the main contributions of our work as follows:

- we bridge the gap between Modelica-based tools and conventional domain-specific power system tools;
- we automate the process of providing good initial guess values to solve the initialization process of Modelica-based power system models with the OpenIPSL library;
- we make Modelica-based tools more attractive for dynamic power system simulation, making it easier for users to use OpenIPSL models for analysis under multiple operating conditions by a power flow record structure;
- with the contributions above, we facilitate the potential adoption of and transition to Modelica-based tools by power system domain specialists.

This chapter is structured as follows: Section 3.2 gives a brief introduction to the power flow problem in electrical networks. In Section 3.3, we introduce the records structure proposed to handle power flow variables. We illustrate how this data container can be linked to OpenIPSL models in Section 3.4, where we also benchmark the power flow values against the results obtained with commercial tools. Finally, Section 3.5 concludes the work.

3.2 The Power Flow Problem

The *power flow* (historically referred to as *load flow*) problem is undoubtedly one of the most performed calculations in power system applications [33]. For instance, these calculations are carried out many times in operations and planning procedures for power grids. An application of a particular interest to this chapter is that power flow solutions provide a *potential* starting guess to solve the initialization problem at which a dynamic simulation may start. Hence, a power flow can be considered one of the most critical problems to be solved when studying a power system [34].

The problem, however, is not new, and neither are the techniques used to solve it. The first practical solutions began to appear in the mid-1950s with the aid of digital computers [35] and a breakthrough came about a decade later. The development of incredibly efficient handling of sparse matrices [36] was paramount to the wide adoption of Newton-Raphson (NR) algorithm. As new issues to solve the power flow problem have arisen, a myriad of new techniques and methods have been proposed; however, NR-based techniques are still the most preeminent methods [33], [34].

From the mathematical perspective, a power flow problem is posed as a set of nonlinear algebraic equations. Its solution will determine an operational point for a specified loading and generation condition in the power system. This operational point is defined by the voltage magnitude and angle in each bus of the system, together with the active and reactive powers generated and consumed in generation and load buses respectively. In the current chapter, we will give a brief introduction to its formulation.

In most power flow formulations, the power system is assumed to be perfectly balanced and operating at a constant frequency (i.e. 50/60 Hz) which would allow it to be represented in its positive sequence equivalent circuit [33]. If a system can be represented in its positive sequence equivalent, it is then possible to assemble its nodal admittance matrix $\mathbf{Y}$ and to write the *nodal equation* as follows:

$$\bar{\mathbf{I}} = \mathbf{Y}\bar{\mathbf{V}}, \tag{3.1}$$

where $\bar{\mathbf{I}}$ is the nodal injection current phasor vector, $\bar{\mathbf{V}}$ is the nodal voltage phasor vector and $\mathbf{Y}$ is the admittance matrix, which is square and sparse.

We could use the nodal equation to compute the voltage at all nodes if all current injection measurements were available. Unfortunately, this is not the case in an electric grid where the known quantities differ from bus to bus. For instance, in a load node, active and reactive power consumption (P, Q) are assumed to be known. Likewise, in a generation bus, active power injection and voltage magnitude at the generator terminals are typically known (P, V). Then, the nodal equation has to be reformulated in terms of P, Q, and V. Because the steady-state relationship between power and voltage/current is nonlinear (and complex-valued), the linear nodal equation into a nonlinear set of complex-valued equations on P, Q, V, and the voltage phasor angle θ.

The exact formulation in the power system jargon is the following. For the mth bus, four variables are either specified or should be calculated in a power flow: active power injected in the bus in per unit P_m, reactive power injected in the bus in per unit Q_m, node voltage phasor magnitude in per unit V_m and node voltage phasor angle in radians θ_m. In load buses (identified as PQ buses) the variable that is known beforehand is the specified apparent power ($S_m^{\mathrm{sp}} = P_m^{\mathrm{sp}} + jQ_m^{\mathrm{sp}}$), while in generation buses (called PV buses) the specified variables are the voltage magnitude ($\bar{V}_m$) and active power (P_m^{sp}). In addition to that, there should be one *slack bus* which should have a specified voltage magnitude value and, most importantly, it should be responsible for providing the angle reference used for all other calculations [33].

The power flow solution is achieved when the computation of active and reactive power via the nodal equation, using the solution values from the most recent iteration, matches the given data for active and reactive power. In other words, the solution occurs when the mismatch between specified and calculated power values is less than or equal to some given tolerance. We can write such a mismatch as

$$\Delta S_m = S_m^{\mathrm{sp}} - V_m I_m^* = P_m^{\mathrm{sp}} + jQ_m^{\mathrm{sp}} - \bar{V}_m \sum_{k \in \mathcal{K}_m} Y_{mk}^* \bar{V}_k^*, \tag{3.2}$$

where $\mathcal{K}_m$ is the set of buses k which are directly connected to bus m and the superscript (*) denotes the complex conjugate. By using the fact that $Y_{mk} = G_{mk} + jB_{mk}$ and expressing the phasor $\bar{V}_m$ as $\bar{V}_m = V_m(\cos\theta_m + j\sin\theta_m)$, we can split Equation (3.2) into its real and imaginary parts as:

$$\begin{cases} \Delta P_m = P_m^{\mathrm{sp}} - V_m \sum_{k \in \mathcal{K}_m} \left(G_{mk}\cos\theta_{mk} + B_{mk}\sin\theta_{mk}\right) V_k, \\ \Delta Q_m = Q_m^{\mathrm{sp}} - V_m \sum_{k \in \mathcal{K}_m} \left(G_{mk}\sin\theta_{mk} - B_{mk}\cos\theta_{mk}\right) V_k, \end{cases}$$

where $\theta_{mk} = \theta_m - \theta_k$. Note that, in this polar formulation, the unknown variables are the nodal voltage phasor magnitudes (V_m) and angles (θ_m).

As said previously, a power flow problem is typically solved using an NR algorithm. First, a nonlinear vector function $\mathbf{f} : \mathbb{R}^{2n} \mapsto \mathbb{R}^{2n}$ is defined, where n is the total number of nodes (buses). This function could be expressed as:

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \Delta\mathbf{P} \\ \Delta\mathbf{Q} \end{bmatrix}, \tag{3.3}$$

where the $\Delta\mathbf{P}$ and $\Delta\mathbf{Q}$ are the n-row vectors $[\Delta P_m]$ and $[\Delta Q_m]$, respectively. In addition,

$\mathbf{x} \in \mathbb{R}^{2n}$ is given by

$$\mathbf{x} = \begin{bmatrix} V_1 & \cdots & V_m & \cdots & V_n & \theta_1 & \cdots & \theta_m & \cdots & \theta_n \end{bmatrix}^T, \tag{3.4}$$

where the superscript ^{T} stands for transpose. To find the power flow solution, we state the following equation [34].

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{3.5}$$

Due to the nonlinear nature of $\mathbf{f}$, it is very difficult to find a closed-form solution for such a problem. Therefore, it is necessary to use an iterative method such as a classical NR algorithm. The i-th iteration of the NR method is written as [34]

$$\begin{cases} \Delta\mathbf{x}_i = [\mathbf{J}(\mathbf{x}_i)]^{-1}\mathbf{f}(\mathbf{x}_i), \\ \mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i, \end{cases} \tag{3.6}$$

where $\mathbf{J}(\mathbf{x})$ is the Jacobian matrix of function $\mathbf{f}$. The iterative method will stop when $\mathbf{f}(\mathbf{x})$ is sufficiently close to $\mathbf{0}$ or, in other words, when its norm is less than a tolerance set by the user. Besides, there are many different ways to find $\mathbf{x}_0$, which is used to start the process described in (3.6). Generally, robust techniques usually allow finding a solution when using a flat start, i.e., when all voltage magnitudes are started as 1 per unit and all angles are started as 0 radians, or one could use a previous power flow solution can be used for the same system.

3.3 Power Flow Records Structure

One of the Modelica language's main advantages is the object-oriented paradigm that enables the user to create dynamic system models hierarchically. Such a structure allows the user to manage model parameters systematically.

A Modelica `record` is a data container used to store a wide range of information about a model, such as parameter values, simulation settings, or values of specific variables

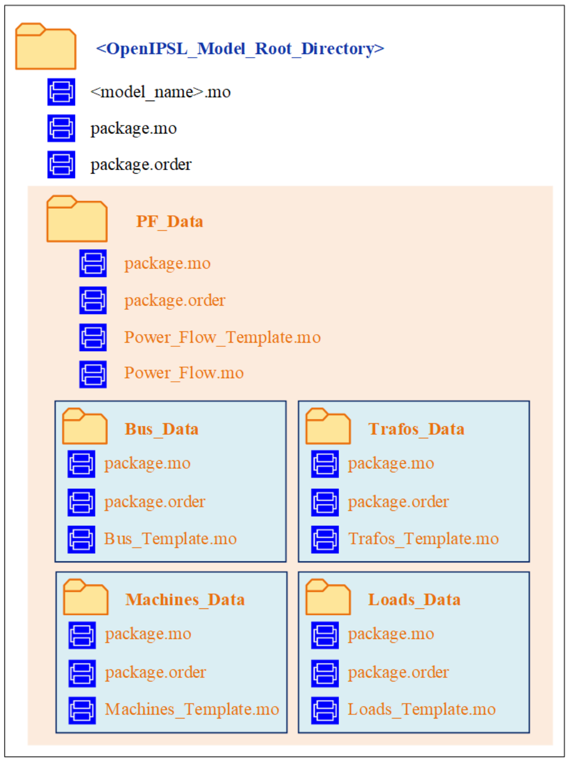


Figure 3.1: File structure of the power flow records inside the OpenIPSL model directory.

for several analysis conditions. Records permit changing a significant number of variables of a given model by modifying just one parameter that related hierarchically to many variables inside the data representation.

Records are a perfect structure for handling power flow values in a dynamic simulation. Suppose power flow results are handled using a record-based data structure. In that case, we could modify the power flow condition of an OpenIPSL model by varying only one attribute of the model, namely, the power flow record, rather than individually changing multiple variables.

The proposed power flow record structure is presented in Figure 3.1. A Python script called `create_pf_records` creates the Modelica files containing the `record` structure and places them inside the model's root folder in a directory called `PF_Data`. This function reads the `.mo` file of the model (`<model_name>.mo`) as a plain text file and uses several regular expressions to determine the number of buses, generators, loads, and transformers in the network. Such a script becomes handy when a user has an existing OpenIPSL model and would like to add a power flow records structure automatically.

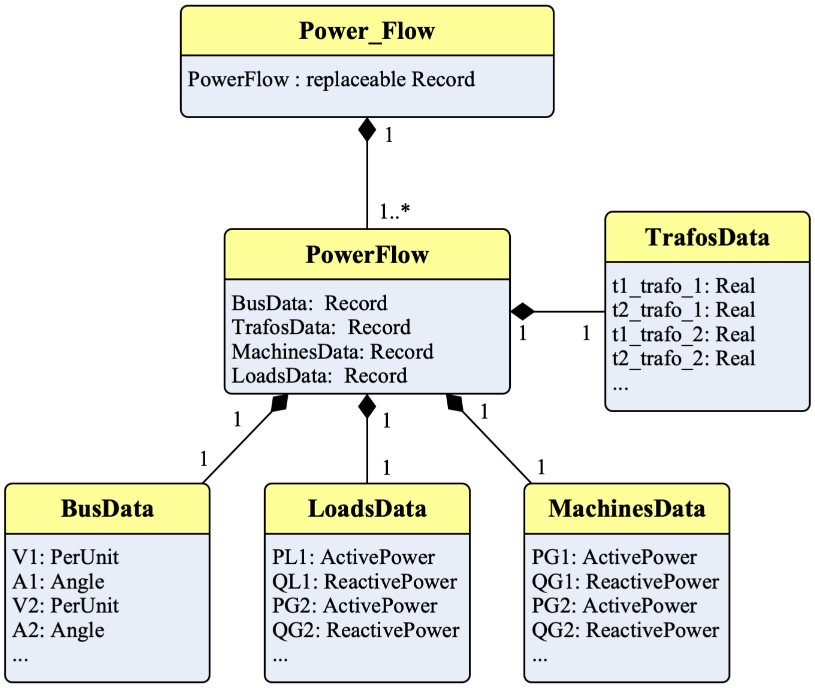


Figure 3.2: Class diagram for the proposed power flow record structure.

We define our power flow record structure in the class `Power_Flow` shown in Figure 3.2. `Power_Flow` is a record having a single attribute: a `replaceable record PowerFlow` . A replaceable condition allows the user having many power flow results for the same model (see Figure 3.4a).

The main idea behind the proposed nested structure is that, by setting the value of `PowerFlow`, the user changes all the power flow variables at once. So, a model has a unique `Power_Flow` record whose power flow attribute is replaceable.

`PowerFlow` has four attributes, which are also records themselves: a record for bus voltages and angles (`Bus_Data`), another for transformer tap positions (`Trafos_Data`), a third one for active and reactive power consumption (`Load_Data`), and a fourth record for machine power dispatch (`Machines_Data`). Naturally, the number of variables inside each of these internal records depends on each particular power system model. For each record type, the variables are specified by the `partial record` templates called `Bus_Template`, `Trafos_Template`, `Machines_Template`, and `Loads_Template`, respectively.

The numerical results are written by a parser function that translates the power flow result from a GridCal model computation into a format compatible with the Modelica record structure. This function is called `gridcal2rec`. `gridcal2rec` creates a `PowerFlow` instance

Source Code 3.1: Creation of the records structure.

```python
from pf2rec import *
import os

# Current working directory
_wd = os.getcwd()
_model_package = 'SMIB'

# Path to the model package directory
data_path = os.path.join(_wd, _model_package)
data_path = os.path.abspath(data_path)

path_mo = os.path.join(data_path,
    'SMIB_Base_case.mo')
path_mo = os.path.abspath(path_mo)

# Creating records structure
create_pf_records(_model_package, path_mo,
    data_path,
    openipsl_version = '2.0.0')
```

placed inside `PF_Data`, whose attributes are four record instances: `Bus_Data`, `Trafos_Data`, `Machines_Data`, and `Loads_Data`.

3.4 Computing and Linking PF Records

This section describes how the records structure, illustrated previously, can be successfully applied to grid models of different sizes.

3.4.1 Creation of Records Structure

A user can integrate our proposed power flow structure into any existing OpenIPSL model using the code contained within the `pf2rec` library (available on GitHub). The records structure is instantiated by the `create_pf_records` function. Listing 3.1 presents a minimal example of creating a power flow record for the Single Machine Infinite Bus (SMIB) system.

Note that the content of the Modelica model is saved within several `*.mo` files. This practice is encouraged since it allows to increase the complexity of the data layer of the model.

As supplementary material to this chapter, we provide a tutorial[4] showing the step-by-step construction of the SMIB system, the corresponding power flow records integration, and computation using GridCal.

The four arguments that we pass to `create_pf_records` are the name of the containing package (`_model_package`), the path to the `*.mo` file where the model is declared (`path_mo`), the path to the containing folder of the model package (`data_path`), and the OpenIPSL library version on which the model has been developed (`_version`). Here, the paths are constructed as absolute references thanks to the `os` library. Such a workaround is recommended to avoid any path problems since the records instantiation involves file/folder creation. The script places all the power flow record `*.mo` files inside a new directory called `PF_Data`. `PF_Data` is also added to the `package.order` file of the root package. In this way, the `record` structure is loaded with the model automatically. Once the power flow is created, the data structure in Figure 3.1 is shown as a nested subpackage, illustrated in Figure 3.3.

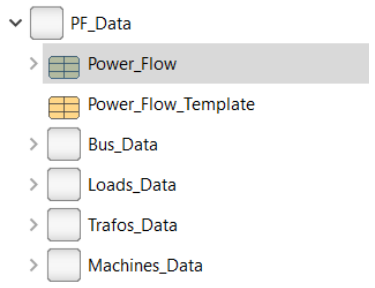


Figure 3.3: Power flow record structure as a nested subpackage in the model structure.

3.4.2 Power Flow Computation with GridCal

GridCal is a Python-based object-oriented software for the computation of power flow results. An example of using GridCal to compute power flow and Python to write the power flow solution into record is shown in Listing 3.2. In this case, a PSS®E `*.raw` file containing the static model information is translated into a GridCal object using the built-in parser class `FileOpen`. The `*.raw` contains the static model of the network, which is required for any power flow formulation. The `*.raw` parser allows us to benchmark the performance of GridCal against PSS®E in terms of power flow result accuracy. Furthermore,

[4]https://github.com/ALSETLab/SMIB_Tutorial/ and https://youtu.be/4qfKw9SAXFY.

this feature reduces the cost of migrating a model from PSS®E to OpenIPSL since the user could initialize both models from the same `*.raw` file. However, the user can define their own grid models from scratch. The reader is referred to the GridCal documentation for network implementation examples.

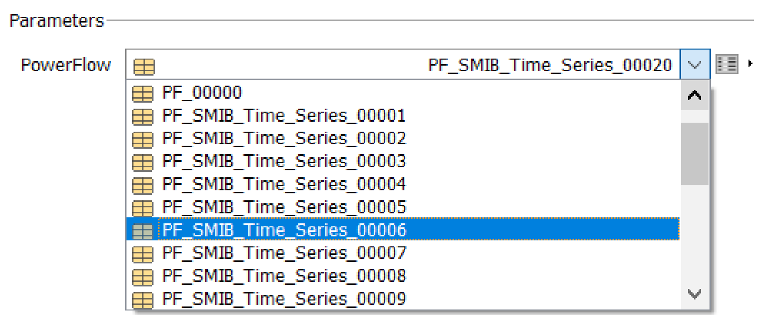
After creating the grid object via the parser class, an instance of the `PowerFlowDriver` is declared: `pf`. `pf` is responsible for carrying out the power flow computation following user-specified settings (`options`). Recall from Equation (3.6) that the method for a power flow computation is constrained by the grid topology (i.e., the matrix $\mathbf{J}(\mathbf{x})$). Therefore, the `grid` object must be passed to the `PowerFlowDriver` constructor method for any power flow computation. The PSS®E `*.raw` file can store up to one power flow result. We take advantage of this fact and use that power flow as an initialization value for a base-case power flow computation in GridCal. The result of this base case should be the same power flow (within the solver's tolerance) as the one included in the PSS®E file. The power flow calculation is commanded by invoking the function `pf.run()`. The results are stored as an attribute of the `PowerFlowDriver` class.

Finally, the function `gridcal2rec` takes the grid information and the power flow driver information and writes the results as Modelica records, following the structure described in Section 3.3. The new files are placed within the `PF_Data` subfolder, housing the power flow record structure. They are also written automatically inside the corresponding `package.order` file to become available to the user right after the computation is completed. The function `gridcal2rec` can be included in automation loops to perform a time series power flow. The resulting output is shown in Figure 3.4a.

In Figure 3.4b one can notice how the power flow condition, defining several variables in a model, can be set either from the graphical interface or by redeclaring a single parameter in the text layer. To the authors' best knowledge, such a feature is not typically available in commercial power system software for dynamics. However, we can easily incorporate it into OpenIPSL models by exploiting the flexibility of object-oriented structure of the Modelica language.

Source Code 3.2: Generation of power flow result using GridCal (PSS®E file input).

```python
_wd = os.getcwd() # working directory
_model_package = 'SMIB'

# Path to the model package directory
data_path = os.path.join(_wd, _model_package)
data_path = os.path.abspath(data_path)

# Path to the PSSE `.raw` file
psse_raw_path = os.path.join(data_path, "PSSE_Files",
    "SMIB_Base_Case.raw")
psse_raw_path = os.path.abspath(psse_raw_path)

# Grid model in GridCal
file_handler = FileOpen(psse_raw_path)

# Creating grid object and setting options
grid = file_handler.open()
options = PowerFlowOptions(SolverType.NR,
    verbose = True,
    initialize_with_existing_solution = False,
    multi_core = False,
    tolerance = 1e-6,
    max_iter = 99,
    control_q = ReactivePowerControlMode.Direct)

pf = PowerFlowDriver(grid, options)
pf.run()

# Writing power flow results in records
gridcal2rec(grid = grid, pf = pf, model_name = 'SMIB',
    data_path = data_path,
    pf_num = 0,
    export_pf_results = False)
```

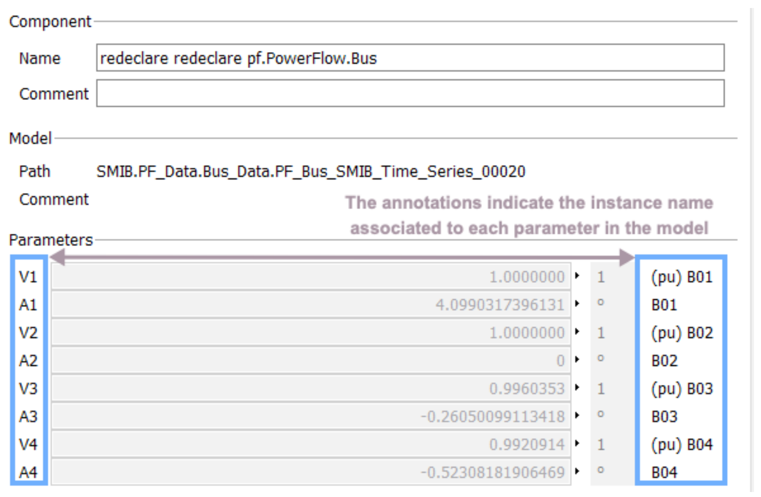


Parameters

PowerFlow PF_SMIB_Time_Series_00020

- PF_00000
- PF_SMIB_Time_Series_00001
- PF_SMIB_Time_Series_00002
- PF_SMIB_Time_Series_00003
- PF_SMIB_Time_Series_00004
- PF_SMIB_Time_Series_00005
- PF_SMIB_Time_Series_00006
- PF_SMIB_Time_Series_00007
- PF_SMIB_Time_Series_00008
- PF_SMIB_Time_Series_00009

(a) Multiple power flows within an OpenIPSL model.



Component

Name: redeclare redeclare pf.PowerFlow.Bus

Comment:

Model

Path: SMIB.PF_Data.Bus_Data.PF_Bus_SMIB_Time_Series_00020

Comment:

The annotations indicate the instance name associated to each parameter in the model

Parameters

V1	1.0000000	1	(pu) B01
A1	4.0990317396131	°	B01
V2	1.0000000	1	(pu) B02
A2	0	°	B02
V3	0.9960353	1	(pu) B03
A3	-0.26050099113418	°	B03
V4	0.9920914	1	(pu) B04
A4	-0.52308181906469	°	B04

(b) Informative annotations to assist the user link the record attributes to the model correctly.

Figure 3.4: Graphical interface of the records structure in Dymola.

A possible difficulty of our power flow generation tool is that the user must connect the power flow parameters in each device to the record manually. After several attempts, we noticed that it depended on how the user constructed a particular model, which is unpredictable. However, we included informative annotations in the record attributes to link the initialization values (see Figure 3.4b) correctly.

Despite this caveat, referencing of the power flow variables to the record must be done only *once*. Afterwards, the user must change the Powerflow attribute, not the record itself. Since the references point to the record on the top layer, they remain unchanged. A detailed example of this process in the tutorial[5] accompanying this chapter.

[5]https://youtu.be/RMD8WEOi6r4.

3.4.3 Scalability for Larger Models

We validated our approach in several systems of different number of buses (that defines the scale of the power flow problem) and of state variables (that defines the size of the complexity of dynamic simulation problem). Table 3.1 and Figure 3.5 summarize the characteristics of the benchmarked systems and illustrate the tool's performance in terms of execution time for record generation and power flow computation. The results correspond to the best scenario over 100 repetitions with 100 loops each. All models are available within the Application Examples of OpenIPSL.

Table 3.1: Scalability results on different systems.

System (Buses)	Number of Variables		Avg. Execution Time (over 100 loops)	
	Algebraic	State	Record	Power Flow Computation
SMIB (4)	99	9	4.08 ms ± 255 μs	31.6 ms ± 1 ms
IEEE 9 (9)	241	29	7.29 ms ± 287 μs	35.5 ms ± 1.55 ms
Kundur Two Areas (11)	244	20	5.07 ms ± 194 μs	37.4 ms ± 1.01 ms
AVRI (14)	16	233	5.77 ms ± 144 μs	35.9 ms ± 1.17 ms
Nordic 44 (44)	1294	6315	55.2ms ± 874 μs	349 ms ± 12.8 ms

The Record creation (RC) process is 5–7x faster than the power flow computation, as expected[6]. Both procedures scale up with the number of algebraic variables, directly related to the dimensionality of the power flow problem. Notice that increase in execution time to generate the records shows an exponential trend with respect to the size of the power flow problem (Figure 3.6), as expected.

[6]The experiments were performed on an Intel Core i5 Quad-Core (2.0 GHz) processor, with 16 GB RAM DDR4 memory.

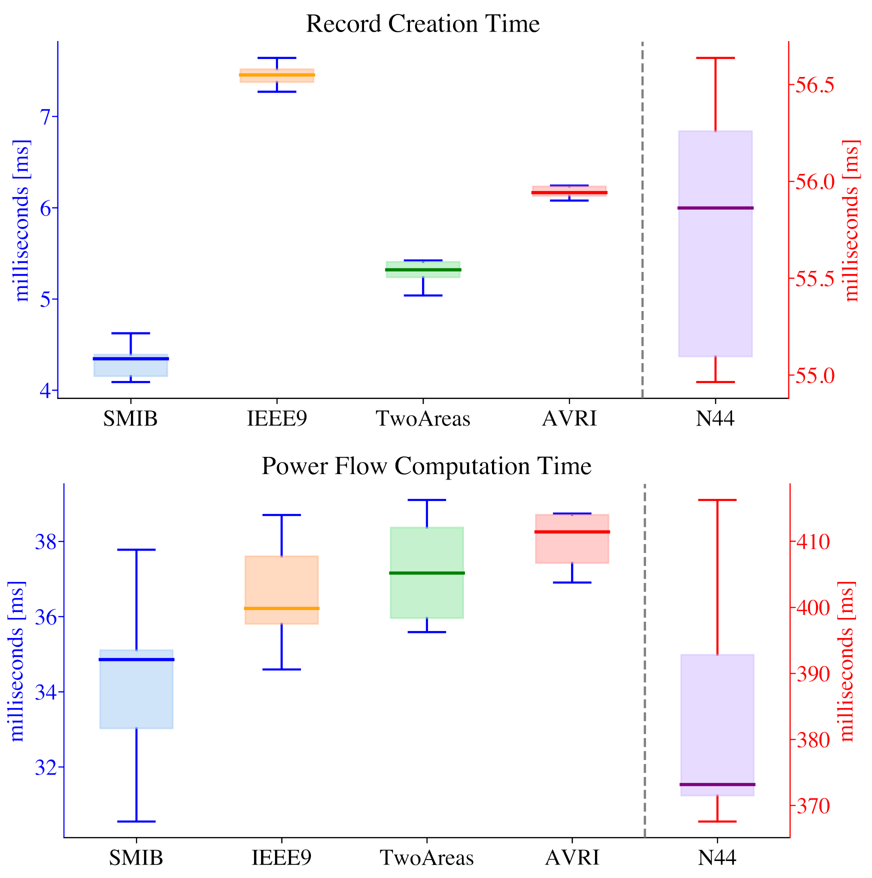


Figure 3.5: Execution time for record creation and power flow computation. Observe that the results for the N44 are presented on a different scale.

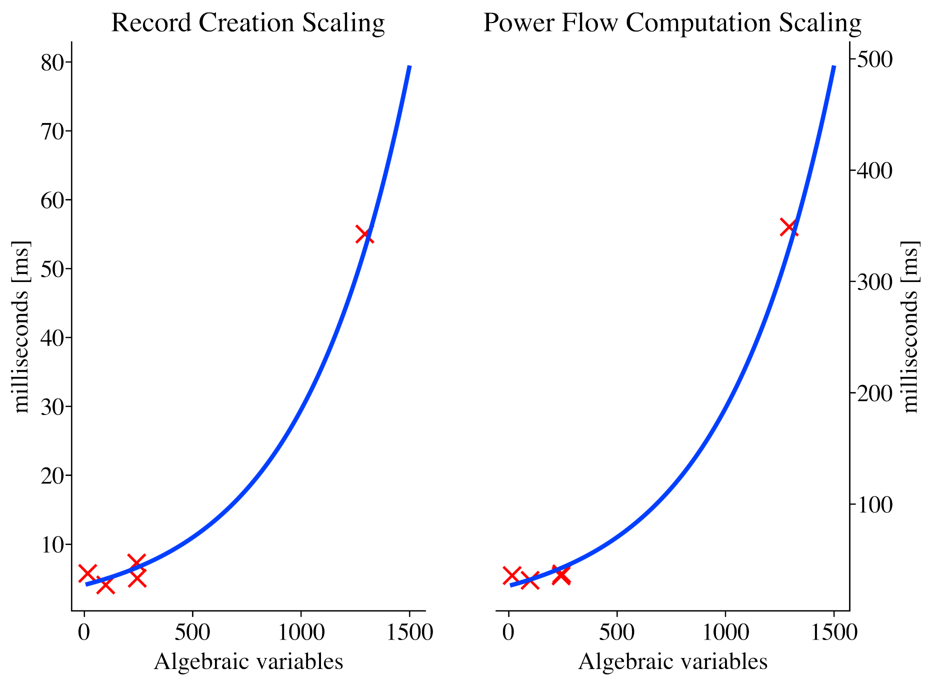


Figure 3.6: Exponential increase in execution time as a function of the number of algebraic variables in the model for the RC.

Table 3.2: Power Flow comparison between PSS®E and GridCal.

System	Bus	Voltage						Power					
		Magnitude [pu]		Absolute Error	Angle [deg]		Absolute Error	P [MW]		Absolute Error	Q [MVar]		Absolute Error
		PSS®E	GridCal		PSS®E	GridCal		PSS®E	GridCal		PSS®E	GridCal	
SMIB	1	1.0000	1.0000	-9.99×10^{-16}	4.04628	4.04627	-2.24×10^{-6}	40.000	40.000	0.00000	5.417	5.417	3.66×10^{-8}
	2	1.0000	1.0000	0.00000	0.00000	0.00000	0.00000	10.017	10.017	5.63×10^{-6}	8.007	8.007	3.83×10^{-8}
IEEE9	1	1.0400	1.0400	0.00000	0.00000	0.00000	0.00000	71.613	71.613	1.11×10^{-5}	25.592	25.592	4.12×10^{-6}
	2	1.0300	1.0300	0.00000	9.18220	9.18219	-4.36×10^{-6}	163.000	163.000	0.00000	8.925	8.925	-3.69×10^{-6}
	3	1.0250	1.0250	0.00000	4.64766	4.64766	-2.20×10^{-6}	85.000	85.000	0.00000	-12.503	-12.503	-1.23×10^{-5}
Two Areas	1	1.0300	1.0300	0.00000	27.07087	27.07086	-7.19×10^{-6}	700.000	700.000	0.00000	185.035	185.035	-2.56×10^{-5}
	2	1.0100	1.0100	0.00000	17.30648	17.30647	-7.33×10^{-6}	700.000	700.000	0.00000	234.624	234.624	-2.10×10^{-5}
	3	1.0300	1.0300	0.00000	0.00000	0.00000	0.00000	719.095	719.095	-2.58×10^{-5}	176.040	176.040	2.24×10^{-5}
	4	1.0100	1.0100	-9.99×10^{-15}	-10.19216	-10.19215	1.09×10^{-5}	700.000	700.000	0.00000	202.114	202.114	-4.49×10^{-5}
AVRI	1	1.0500	1.0500	0.00000	0.00000	0.00000	0.00000	-100.000	-100.000	0.00000	41.391	41.391	-4.25×10^{-6}
	8	1.0500	1.0500	0.00000	47.01978	47.01976	-1.89×10^{-5}	50.000	50.000	0.00000	19.795	19.795	-7.89×10^{-7}
	12	1.0500	1.0500	0.00000	43.26172	43.26170	-2.11×10^{-5}	50.000	50.000	0.00000	21.916	21.916	-4.04×10^{-6}
N44	3115	1.0000	1.0000	0.00000	-13.59220	-13.59220	1.12×10^{-6}	1114.875	1114.875	0.00000	-395.702	-395.702	1.37×10^{-5}
	6000	1.0050	1.0050	0.00000	-18.37864	-18.37864	-2.86×10^{-6}	1010.808	1010.808	0.00000	-400.800	-400.780	1.97×10^{-2}
	6500	1.0000	1.0000	0.00000	-25.88593	-25.88593	-3.62×10^{-6}	1093.284	1093.284	0.00000	882.375	882.375	-1.36×10^{-4}
	8500	1.0200	1.0200	-9.99×10^{-15}	-5.72443	-5.72443	5.95×10^{-7}	1952.664	1952.664	0.00000	596.683	596.683	2.58×10^{-4}

3.4.4 Result Validation with PSS®E

The validation against PSS®E of the power flow results obtained using GridCal has been performed on several test systems. In Table 3.2, a list of the tested networks is given. For each of the networks some buses have been selected indicating their voltage magnitude and angle, the injected/absorbed active and reactive powers of the generating units connected to the corresponding node. Those power flow results are compared with the corresponding calculations obtained from PSS®E including evaluation of an absolute error between the evaluated power flow and reference PSS®E power flow. The power flow values match with low tolerance errors that in some cases hit the machine precision. This shows the validity of the proposed approach of power flow calculation using GridCal.

3.5 Conclusions

This chapter presented an approach to form a record-based data structure to handle power flow starting guesses for a dynamic simulation using the phasor-domain OpenIPSL library. A power flow computation, performed before running a phasor-domain simulation, specifies the starting equilibrium of the nonlinear system simulation. The record class architecture benefits directly from the object-oriented paradigm of the Modelica language, allowing management of *all* power flow variables from a *single* attribute in the model, a feature not common in specialized proprietary power system tools. Such structure can be extrapolated to other open-source Modelica-based power system libraries.

We provide a Python script to create the structure for any existing OpenIPSL model built on versions 1.5.0 or 2.0.0, in this way, naturally expanding capabilities of the library to perform dynamic simulations for different power flow initial conditions. The power flow record instances can be populated by an open-source Python library called GridCal, capable of producing numerically the same results as PSS®E for power flow computations. We also introduce a script to convert the GridCal power flow results to records directly.

From our perspective, the proposed methodology can be useful for users of existing OpenIPSL models, especially for those who study the behavior of the models under different power flow conditions. However, for large scale models the user would have to spend significant time linking the power flow variables to the record. To avoid the aforementioned issue, a model translation tool that translates the information from PSS®E `*.dyr` and `*.raw` files into OpenIPSL `*.mo` models is currently under development. The tool will include the proposed record structure in this chapter by default. In that case, the power flow variables will point to the record automatically. This will be a key advantage in helping power system analysts with the potential adoption and transition to Modelica-based tools.

CHAPTER 4
SCENARIO GENERATION FOR SMALL-SIGNAL STABILITY ASSESSMENT VIA MACHINE LEARNING

4.1 Introduction

Simulation tools are broadly used to gain insight into current and future operating conditions of the grid. Despite this, due to the vast number of variables and the ubiquitous uncertainty of modern networks, the amount of scenarios that need to be considered by a human operator in simulation-based studies is exponentially large. In this situation, a need arises not only to automate the simulation procedure (to massively generate data) but also to simplify data interpretation. For the former, Python-based solutions have been gaining popularity in almost all engineering fields by providing simple automation solutions (see [38] for an application example in power grids).

Regarding data interpretation, ML techniques are powerful statistical methods that can be used, for example, to extract information from large sets of data [39]. In special, DL is a particular family of ML techniques that employ Neural Networks (NNs) as building blocks. Both ML and DL solutions have been recently applied in power systems for the classification of events from Phasor Measurement Unit (PMU) data [40], [41], voltage stability [42]–[44], and dynamic security assessment [45], among others. ML and DL solutions have been recently applied in power systems for the enhancement and evaluation of small-signal stability. For example, the work of [46] performs coordinated tuning of Power System Stabilizer (PSS) parameters using heuristic optimization algorithms. Likewise, in [47] a cuckoo search is employed to find optimal PSS parameters that guarantee small-signal stability. Regarding NNs, in [48] the parameters of a unified power flow controller are tuned via a NN whose weights are optimized using Levenberg-Marquardt optimization.

On the other hand, within several well-established stability techniques, small-signal

Portions of this chapter appear in S. A. Dorado-Rojas, M. de Castro Fernandes, and L. Vanfretti, "Synthetic Training Data Generation for ML-based Small-Signal Stability Assessment," presented at the 2020 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm) [8], and in the corresponding tutorial of the SmartGridComm conference [37].

analysis quantifies the effects of small disturbances in a given power system. Such small-scale perturbations can lead to large instabilities if specific modes of the system are excited. Traditionally, oscillations are studied by obtaining a linear model of the power system around a stable equilibrium point and evaluating the eigenvalues of this model. Alternatively, eigenvalues can be determined from time-domain measurements or simulation data via traditional signal processing and system identification algorithms [49].

Once the eigenvalues describing a particular small-signal scenario are available, its classification in pre-established categories is straightforward. In fact, given the set of dominant eigenvalues λ_i of the system in a particular contingency scenario, the operational state may be assessed by computing the damping ratio ζ_i for each eigenvalue λ_i. Hence, ζ represents a metric that can be used to define a 100%-accurate classifier to categorize contingency scenarios (ζ-classifier).

Furthermore, the most challenging step from the computational point of view is the state matrix computation rather than the damping ratio calculation, especially if the former is done numerically instead of analytically [50]. Despite this, a system identification-based method is preferred when working with measurements, as it will be more accurate at evaluating the system's condition. A valid question, however, would be if an alternative ML solution could be used to bypass the system identification step while producing more computationally efficient solutions. For this alternative to be practically significant, the ML solution should be accurate enough at both learning the linear system representation and evaluating the system condition from eigenvalues. This chapter focuses on the latter issue and explores the small-signal stability assessment accuracy of several ML techniques.

This chapter takes the challenge to generate massively contingency data and to automate small-signal stability computation by ML methods. The contribution of this work is as follows:

- we propose a simple ad-hoc Monte Carlo sampling technique to generate numerous contingency scenarios for a given power system model. Then, each scenario is simulated in a Modelica-based environment to obtain labeled big data for ML training;
- we use the generated big data to train several conventional ML algorithms (logistic and softmax regression, support vector machines, k-nearest Neighbors, Naïve Bayes and decision trees), and a deep learning NN to classify the operating condition of a

power system after a contingency (i.e., one or more line trips) based on a small-signal stability metric;

- we propose an evaluation metric as a guideline for the selection of the best classifier in terms of its performance.

We verify that once trained, the ML approaches produce results as accurate as the damping ratio-based classifier, which is implemented in Python using `NumPy` in a vectorized fashion (ζ classifier). Moreover, almost all ML solutions take a lower amount of prediction time than the hard-coded domain-specific algorithm to classify operation scenarios. This is desirable for deploying a trained solution inside an online or near real-time application. In particular, the trained NN shows 120x faster prediction time than the damping ratio classifier with an accuracy above 95%.

An important contribution of the chapter concerns the approach for massive data generation. The information required for a linear analysis (that is, the $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ matrices of a linear state-space representation) is obtained through a routine implemented in Dymola, a Modelica-compliant modeling and simulation environment. We take advantage of this built-in functionality to linearize a nonlinear dynamic power system model around an equilibrium point. By doing so, it is possible to perform a small-signal analysis for a vast quantity of scenarios in a power system using a model constructed using the OpenIPSL [15], a library for phasor time-domain analysis in Modelica. Each scenario is generated by a two-stage Monte Carlo sampling procedure that takes into account the topology of the system as described in Section 4.3.

By automating Dymola linearization with Python [51], a vast amount of data is generated for several grid conditions with different small disturbances in the form of contingencies. Such an intensive simulation-based data generation approach has been recently used in other power system studies such as transmission planning as well [52]. In this case, the data is employed to train an automatic classifier such as an ML algorithm to evaluate small-signal condition of the system. The complete code used for this work is available on GitHub[1].

This chapter is organized as follows: in Section 4.2 we present a brief overview of small-signal analysis and how eigenvalues can be classified. The test power system and the

[1]https://github.com/ALSETLab/Synthetic_Data_Generation_ML_Small_Signal

data generation procedure are outlined in Section 4.3. Section 4.4 describes and presents the proposed Neural Network architecture, the results of its training procedure, and its performance after being deployed. Finally, Section 4.6 concludes the work.

4.2 Foundations of Small-Signal Analysis

Consider a generic representation of a power system, ignoring discrete events such as limiters or protections, with m inputs, p outputs and n states, whose state-space is described by $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t)$, and $\mathbf{y} = \mathbf{g}(\mathbf{x}, \mathbf{u}, t)$ where $\mathbf{x} \in \mathbb{R}^{n \times 1}$ is the state vector, $\mathbf{u} \in \mathbb{R}^{m \times 1}$ is the vector of m inputs to the system, and $\mathbf{f} \in \mathbb{R}^{n \times 1}$, $\mathbf{g} \in \mathbb{R}^{p \times 1}$ are two nonlinear $\mathcal{C}^{\infty}$ functions. If time dependence is implicit (i.e., time does not appear explicitly in the system equations), we have

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}) \\ \mathbf{y} &= \mathbf{g}(\mathbf{x}, \mathbf{u}). \end{aligned} \tag{4.1}$$

For the system in Eq. (4.1), an equilibrium exists whenever the state derivatives are zero ($\dot{\mathbf{x}} = \mathbf{0}$). At a given equilibrium point $(\mathbf{x}_0, \mathbf{u}_0)$, we have $\mathbf{f}(\mathbf{x}_0, \mathbf{u}_0) = \mathbf{0}$. Now, we analyze the situation where the system is in equilibrium and a small disturbance occurs. The disturbance brings the system to a new state. The dynamics at the new operating point take the form

$$\begin{aligned} \dot{\tilde{\mathbf{x}}} &= \mathbf{f}(\tilde{\mathbf{x}}, \tilde{\mathbf{u}}) = \mathbf{f}(\mathbf{x}_0 + \Delta\mathbf{x}, \mathbf{u}_0 + \Delta\mathbf{u}) \\ \tilde{\mathbf{y}} &= \mathbf{g}(\tilde{\mathbf{x}}, \tilde{\mathbf{u}}) = \mathbf{g}(\mathbf{x}_0 + \Delta\mathbf{x}, \mathbf{u}_0 + \Delta\mathbf{u}). \end{aligned} \tag{4.2}$$

If the disturbance under consideration is small enough[2], we can perform a Taylor Series expansion around an equilibrium point for both functions $\mathbf{f}$ and $\mathbf{g}$ to obtain a linear representation of the nonlinear system. This can be achieved by a first-order Taylor Series truncation neglecting all terms of order larger than one, thus keeping only the matrix of first-order derivatives (Jacobian linearization). An application of this formula to the system in Eq. (4.1) leads us to

$$\begin{aligned} \Delta\dot{\mathbf{x}} &= \mathbf{A}\Delta\mathbf{x} + \mathbf{B}\Delta\mathbf{u} \\ \Delta\mathbf{y} &= \mathbf{C}\Delta\mathbf{x} + \mathbf{D}\Delta\mathbf{u} \end{aligned} \tag{4.3}$$

[2]This means that the region of convergence of the Taylor Series corresponds to a non-empty set in the neighborhood of the equilibrium point.

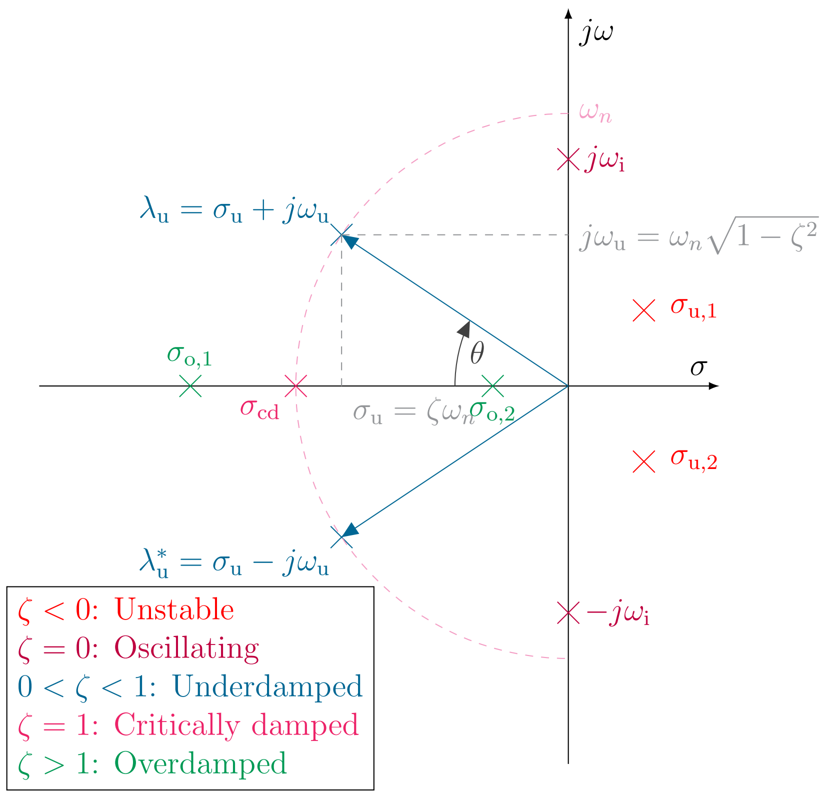

$j\omega$
ω_n
$j\omega_{\mathrm{i}}$
$\lambda_{\mathrm{u}} = \sigma_{\mathrm{u}} + j\omega_{\mathrm{u}}$
$j\omega_{\mathrm{u}} = \omega_n\sqrt{1-\zeta^2}$
$\sigma_{\mathrm{u},1}$
$\sigma_{\mathrm{o},1}$
θ
σ
σ_{cd}
$\sigma_{\mathrm{u}} = \zeta\omega_n$
$\sigma_{\mathrm{o},2}$
$\sigma_{\mathrm{u},2}$
$\lambda_{\mathrm{u}}^{*} = \sigma_{\mathrm{u}} - j\omega_{\mathrm{u}}$
$-j\omega_{\mathrm{i}}$
$\zeta < 0$: Unstable
$\zeta = 0$: Oscillating
$0 < \zeta < 1$: Underdamped
$\zeta = 1$: Critically damped
$\zeta > 1$: Overdamped

Figure 4.1: Damping factor depending on different eigenvalue locations on the complex plane.

where $\mathbf{A} \coloneqq \partial\mathbf{f}/\partial\mathbf{x}$, $\mathbf{B} \coloneqq \partial\mathbf{f}/\partial\mathbf{u}$, $\mathbf{C} \coloneqq \partial\mathbf{g}/\partial\mathbf{x}$ and $\mathbf{D} \coloneqq \partial\mathbf{g}/\partial\mathbf{u}$. Note that the system representation in Eq. (4.3) corresponds to a state-space realization of a Linear Time-Invariant system that can be analyzed using linear methods.

Linear analysis techniques can be employed to quantify system behavior after a small disturbance (hence the name *small-signal*) such as tripping of a given line. System modes of a linear system are completely specified by the eigenvalues of the system matrix $\mathbf{A}$. For the ith eigenvalue λ_i with algebraic multiplicity n_i, the associated n_i modes are $\bar{c}_i t^k e^{\lambda_i t}$ for $k = 0,\ 1, \ldots,\ n_i - 1$, with $\bar{c}_i \in \mathbb{C}$. The characteristic of the mode associated to a single eigenvalue can be completely described by a single metric known as *damping factor* or *damping ratio* ζ. As shown in Figure 4.1[3] , the damping factor determines uniquely the characteristics of the system mode associated with a particular eigenvalue.

[3]Since we are using Lyapunov's first method for the nonlinear system in Eq. (4.1), no conclusion regarding equilibrium stability can be drawn if the $\mathbf{A}$ matrix is not Hurwitz (i.e., $\zeta \leq 0$).

Table 4.1: Damping factor of an eigenvalue and associated response nature [53].

Damping Factor	Response Nature
$\zeta < 0$	System (might be) unstable
$\zeta = 0$	System (might be) oscillatory
$0 < \zeta < 1$	Underdamped Response
$\zeta = 1$	Critically Damped Response
$\zeta > 1$	Overdamped Response

Thanks to the characteristics of the damping factor as a discriminative scalar metric (see Table 4.1), it is possible to categorize eigenvalues based on ζ such that each of the classification groups represents a state of an electric grid if the corresponding eigenvalue is linked to a dominant mode. This automatic labeling can help learn the stability condition of a power system.

4.3 Scenario Sampling for Data Generation

The IEEE 14 bus network is used as a test bench to generate large-scale data for NN training by systematically applying contingencies and collecting simulation data. This system, which is shown again in Figure 4.2, counts with 5 generators, 16 lines and 4 transformers. In summary, there are 20 branch elements in the system, all of them having an impedance in the corresponding power system model.

If the branch impedance value X_i of one of these element models is made large enough, we would have emulated a line opening in the system without actually removing the component. In fact, by letting $X_i \approx 10^{12}$ we do not alter the topology of the grid (and do not change the number of states nor the size of the $\mathbf{A}$ matrix) but we effectively "apply" a contingency to the system which is equal to disconnecting the branch.

Considering that each of the 20 branch elements can be tripped, there exist 20 possible scenarios that can help to evaluate the disconnection of a single element. Likewise, if two branches are opened simultaneously, 190 possible scenarios can be tested by selecting all branches pairwise. In general, letting n being the total number of branches, and k the number of simultaneous disconnections, the amount of possible scenarios with this contingency configuration is given by

$$S_{n,k} = \frac{n!}{(n-k)!k!}. \tag{4.4}$$

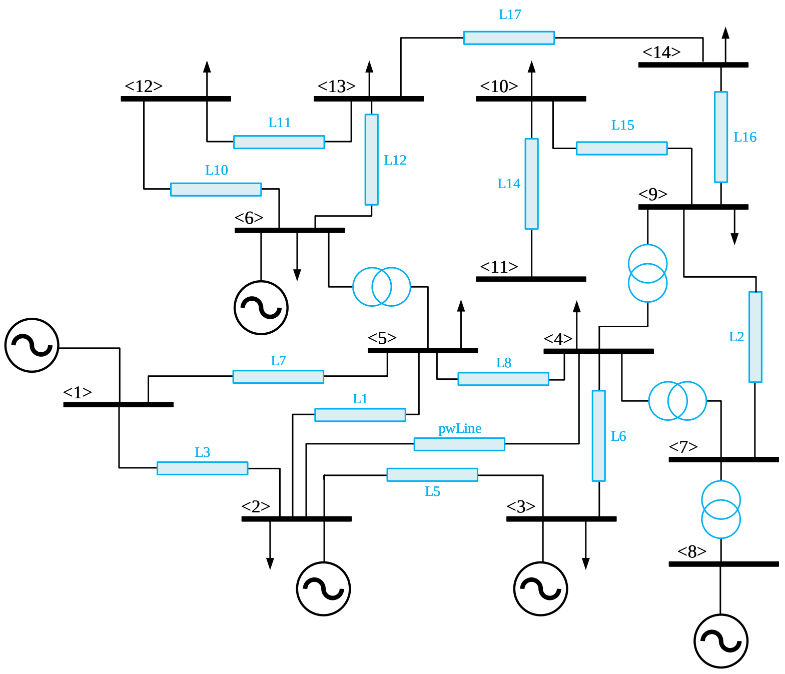


Figure 4.2: IEEE 14 nodes test system.

For this study, there is no need to consider all lines being opened at the same time since it is not physically significant. Hence, a maximum of $k_{\max} = n - 1 = 19$ simultaneous disconnections is considered. In addition, note that $k_{\min} = 1$. Thus, it is possible to calculate the total amount of possible scenarios as

$$T = \sum_{k_{\min}}^{k_{\max}} S_{20,k} = \sum_{k=1}^{19} \frac{20!}{(20-k)!k!} = 1,048,574. \tag{4.5}$$

From the total amount of possible scenarios, T, it is necessary to select a subset of events with physical significance. To address this issue, an ad-hoc Monte Carlo method, consisting of a two-stage sampling procedure to select scenarios, is proposed. In the first stage, the number of lines that will be opened is selected. Here, it is necessary to find a probability distribution that reflects the fact that scenarios with a smaller number of events are more likely to occur and, therefore, should be more likely to be selected. To take this constraint into account, a modified Poisson distribution is proposed, giving larger probabilities to smaller values of k. The Probability Density Function (PDF), is illustrated in Figure 4.3 is defined as:

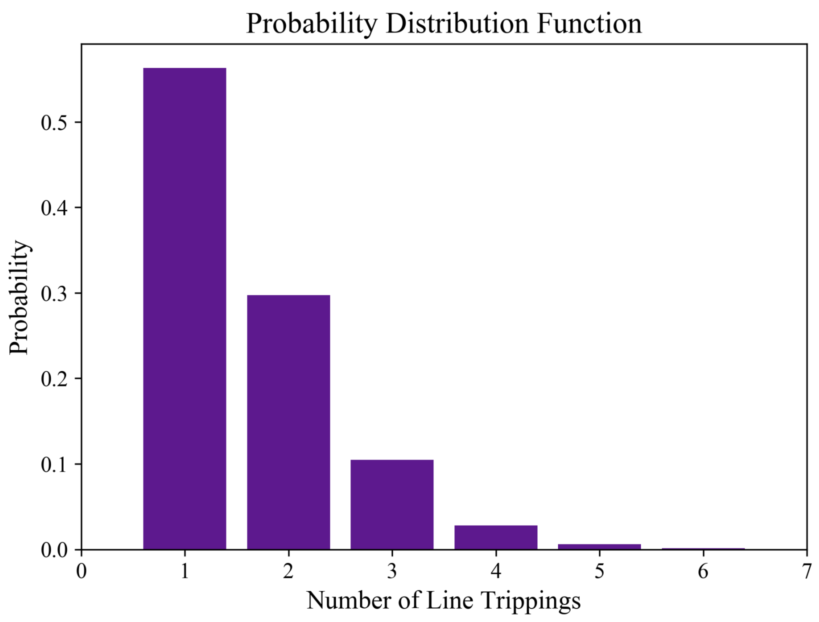


Figure 4.3: Probability distribution function for the number of lines to be opened for contingency generation.

$$p(k) = \frac{1}{k! \sum_{n=1}^{19} \frac{1}{n!}} \approx \frac{1}{k!(e-1)}, \tag{4.6}$$

where e is Euler's number. Once the number of lines is fixed, the second stage starts, and one scenario is selected from the pool of all possible combinations with the specified number of contingencies. Thanks to this ad-hoc method, 20,000 different simulation scenarios are generated.

Once a simulation scenario has been selected, the corresponding branches in the system model in Dymola are disconnected. Dymola's built-in function `linearizeModel` is used to extract the state matrix and the eigenvalues for each scenario. This process is automated in Python using the so-called PDI [51], that allows combining the simulation power of Dymola with Python capabilities for a variety of purposes. The usage of PDI also allows several robust ML development libraries, such as `scikit-learn` and `TensorFlow`, to be used to train classifying ML/DL solutions from the significant amount of data produced by massive simulations.

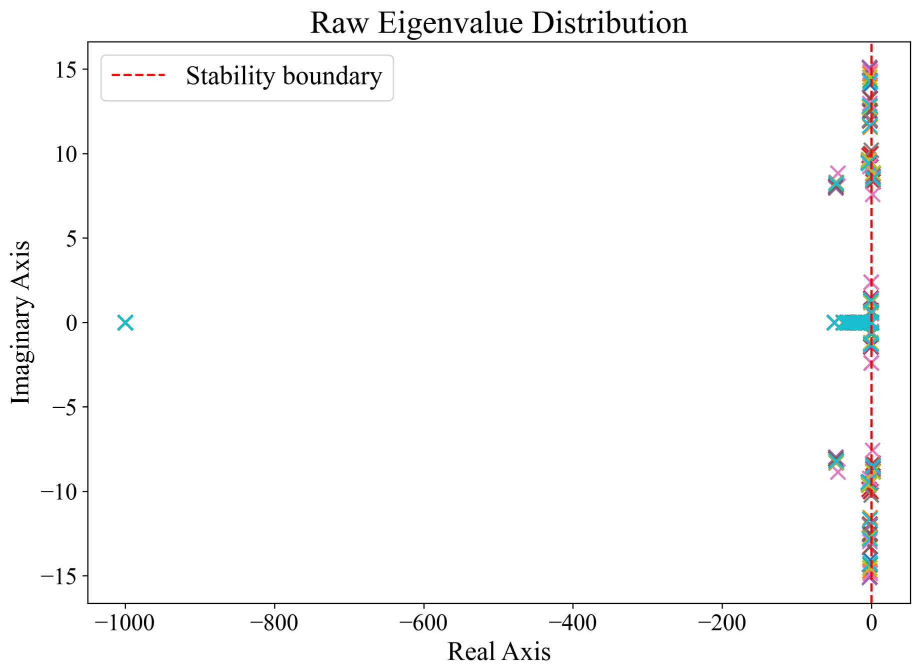


Figure 4.4: Distribution of raw eigenvalues in the complex plane.

4.4 ML Algorithm Design, Training and Performance

In this section, we briefly describe the different steps carried out to design the ML/DL algorithms employed to classify the different contingency scenarios into pre-established categories.

– **Step 1 - Data Generation:** we use 20,000 scenarios of the IEEE 14 bus system to generate eigenvalue data. Dymola succeeded in linearizing 19,815 of those scenarios. Since each one is associated with 49 eigenvalues, a total number of 970,935 eigenvalues was produced. An overview of the generated eigenvalues is shown in Figure 4.4.

– **Step 2 - Data Preprocessing:** the raw data obtained from simulations is organized and labeled (i.e., by manually classifying the eigenvalues, computing the damping ratio and tagging them according to the pre-defined categories below, what we refer to as *hard-coded classifier*). Each eigenvalue is classified within one of six categories (Figure 4.5) that are defined as follows:

1. **Unstable** ($\zeta < 0$): if an eigenvalue lies on the right-half plane.

2. **Stable but critical condition** ($0 \leq \zeta < 0.05$): the eigenvalue is stable or it is oscillatory (so no conclusion can be drawn about the stability of the system). This

is a condition for which an action of the system controls is required since a small disturbance can lead to instabilities and/or heavy oscillatory modes in the system.

3. **Acceptable condition within operating limits** $(0.05 < \zeta < 0.1)$: in this case, the damping of the system is large enough to handle and tolerate a small-disturbance. As a consequence, the operation of the system is labeled as acceptable.

4. **Good operating condition** $(0.1 \leq \zeta < 1)$: for this scenario, the damping ratio is larger than 10% and the response will show some oscillation due to the underdamped nature of the corresponding eigenvalue.

5. **Satisfactory operating condition** $(\zeta > 1.1)$: this category gathers real eigenvalues whose overdamped response is satisfactory in terms of oscillations. Normally, these eigenvalues are not dominant. Hence, they do not impose its dynamics on system response.

6. **Irrelevant (eigenvalue at the origin or close to it)**: category that groups the eigenvalues that are at the origin or close to it (within a neighborhood of radius 0.2). They mostly represent integral relationships between state variables (e.g., between ω and δ).

In the pre-processing stage, the eigenvalues whose magnitude is magnitude larger than one are normalized (Figure 4.6). All λ_i-s lying inside the unit circle are not touched since the information they convey regarding the stability boundary of the system would be lost.
– **Step 3 - Model Design, Training and Evaluation:** we evaluated both classical ML techniques for classification as well as a fully-connected NN. The selected supervised ML algorithms are multi-class logistic regression (LogReg), softmax regression (SoftmaxReg), linear support vector machines (SVM), k-nearest neighbors (k-NN), Naïve Bayes and decision trees.

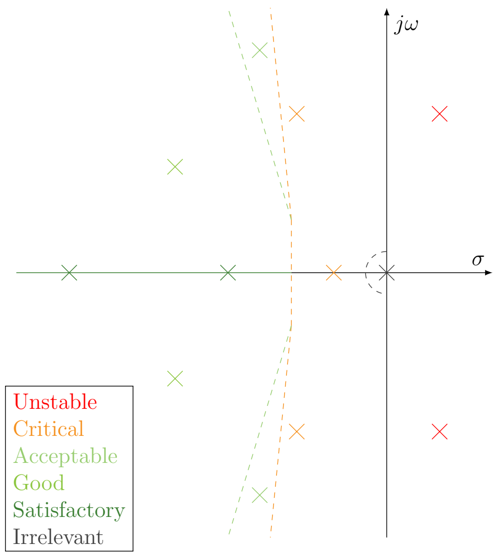


Figure 4.5: Regions of each classification category on the complex plane.

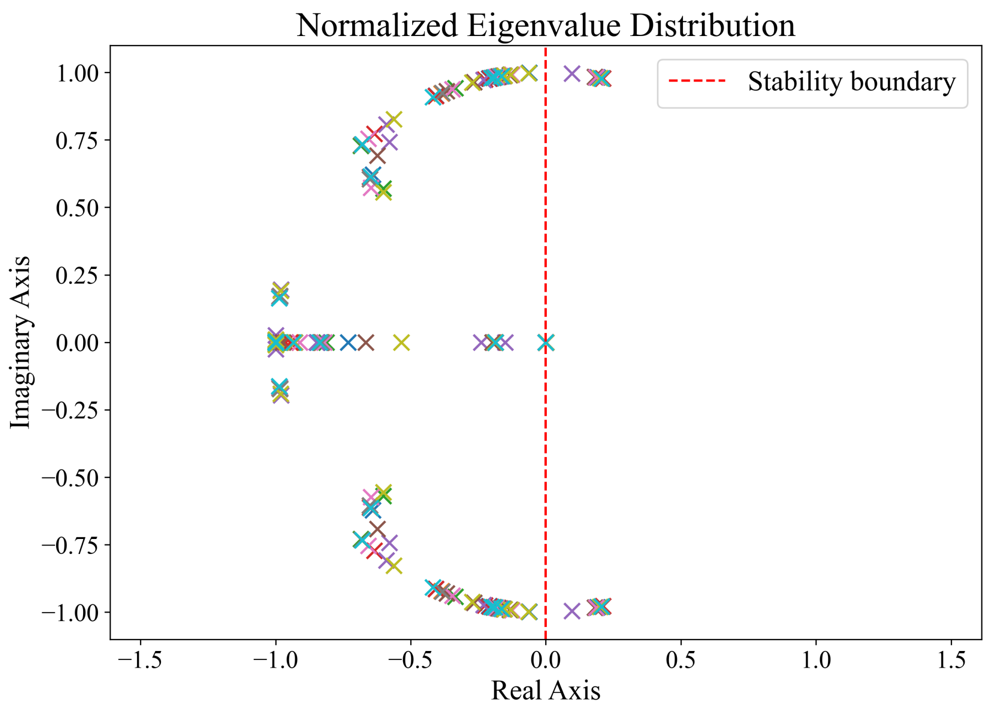


Figure 4.6: Distribution of normalized eigenvalues in the complex plane.

The reader is referred to [39] for an in-depth explanation of each ML algorithm. We will put special attention to the NN design since it is the best performing method among all ML/DL techniques considered in this study.

The proposed NN architecture is presented in Figure 4.7. It consists of six fully-connected layers, four of them with learning parameters. Two dropout layers are added to reduce overfitting and improve generalization performance. The input layer and the two hidden layers employ a rectified linear unit (ReLU) unit as an activation function. Since this is a multi-class classification problem, a softmax function is suitable as an output activation.

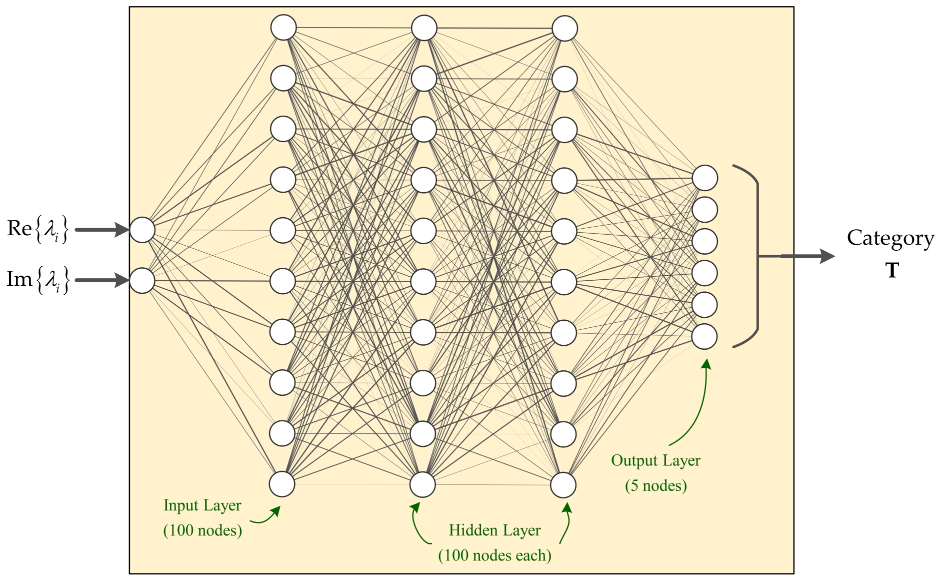


Figure 4.7: NN architecture.

Let m be the number of training samples in a batch. The input feature matrix will be $\mathbf{X}^{m\times 2}$. Then,

$$\begin{aligned}
\mathbf{H}^{[1]} &= \text{ReLU}\left(\mathbf{X}\mathbf{W}^{[1]} + \mathbf{b}^{[1]}\right)\\
\mathbf{D}^{[1]} &= \text{dropout}\left(\mathbf{H}^{[1]}\right)\\
\mathbf{H}^{[2]} &= \text{ReLU}\left(\mathbf{D}^{[1]}\mathbf{W}^{[2]} + \mathbf{b}^{[2]}\right)\\
\mathbf{D}^{[2]} &= \text{dropout}\left(\mathbf{H}^{[2]}\right)\\
\mathbf{H}^{[3]} &= \text{ReLU}\left(\mathbf{D}^{[2]}\mathbf{W}^{[3]} + \mathbf{b}^{[3]}\right)\\
\mathbf{Y} &= \sigma_{\mathcal{M}}\left(\mathbf{H}^{[3]}\mathbf{W}^{[4]} + \mathbf{b}^{[4]}\right)\\
\mathbf{T} &= \text{argmax}\left(\mathbf{Y}\right)
\end{aligned} \tag{4.7}$$

where the hidden states are described by the matrices $\mathbf{H}^{[1]}$, $\mathbf{H}^{[2]}$, $\mathbf{H}^{[3]} \in \mathbb{R}^{m\times 100}$, $\sigma_{\mathcal{M}}\left(\mathbf{Z}^{[4]}\right) \in \mathbb{R}^{m\times 5}$ with $\mathbf{Z}^{[4]} := \mathbf{H}^{[3]}\mathbf{W}^{[4]} + \mathbf{b}^{[4]}$. $\mathbf{D}^{[1]}$ and $\mathbf{D}^{[2]}$ are the outputs of the dropout layers (not trainable). The weights and biases are $\mathbf{W}^{[1]} \in \mathbb{R}^{2\times 100}$, $\mathbf{W}^{[2]} \in \mathbb{R}^{100\times 100}$, $\mathbf{W}^{[3]} \in \mathbb{R}^{100\times 100}$, $\mathbf{W}^{[4]} \in \mathbb{R}^{100\times 5}$, $\mathbf{b}^{[1]}$, $\mathbf{b}^{[2]}$, $\mathbf{b}^{[3]} \in \mathbb{R}^{100\times 1}$, and $\mathbf{b}^{[4]} \in \mathbb{R}^{5\times 1}$. $\mathbf{Y} \in \mathbb{R}^{m\times 5}$ is a matrix whose mth column contains the probability of each of the m inputs belonging to each category. Finally, $\mathbf{T}$ is a matrix whose mth vector has a 1-entry at the position corresponding to the class with the highest probability and zero everywhere else.

The loss function is a cross-entropy function defined minibatch-wise as $\ell_m = -\frac{1}{m}\sum_{i=1}^{5}\log y_i$ where y_i is the ith component of the mth column of $\mathbf{Y}$. Finally, the total loss is computed by adding the individual losses per minibatch as $L = \sum_{<m>} \ell_m$. The weights and biases are learned by minimizing the loss function L. The solution of this complex optimization problem is carried out by the specialized Python library `TensorFlow`. The behavior of the loss function per learning epoch can be detailed in Figure 4.8, together with the results of training and testing accuracy. The number of epochs is set to 50 to get the final values of testing (97.79%) and training (98.60%) accuracies before deploying the NN.

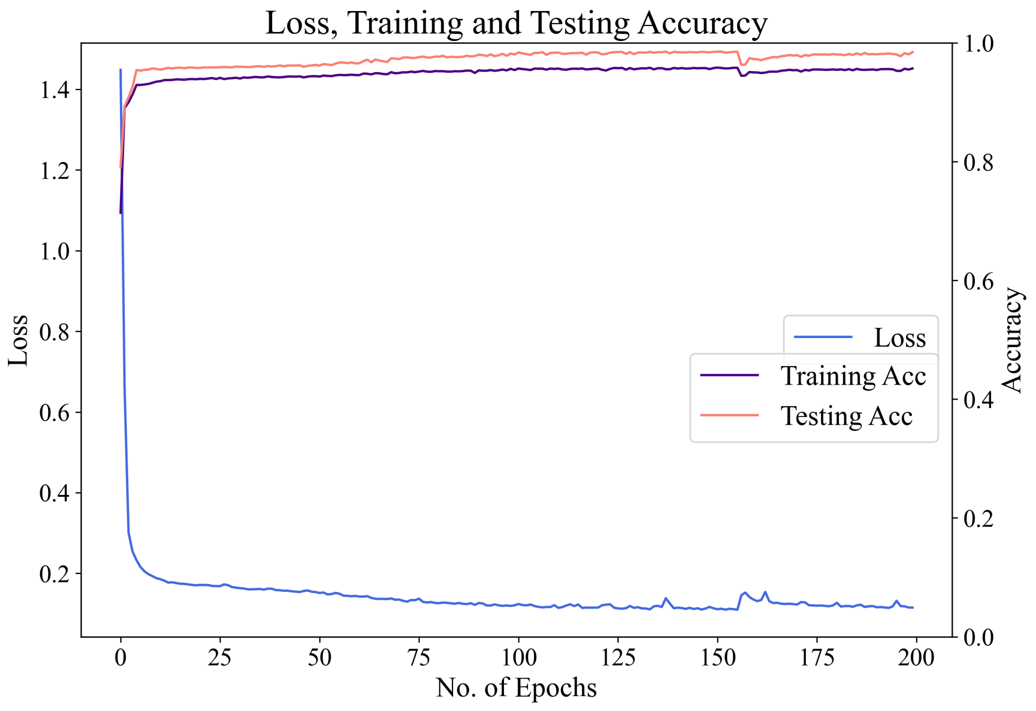


Figure 4.8: Loss function value, training and testing accuracy per training epoch.

Note that the damping ratio computation is required for creating the labels for each category (what we call a hard-coded classifier) but not for the training process. In particular, the NN may learn the damping ratio representation of the eigenvalues in some hidden layers if it is useful for the classification task itself.

Finally, after training the NN is evaluated on the validation set. The results of the prediction for all methods can be found in the GitHub repository, together with more visualization of the predictions. The accuracy on the validation stage for the NN was of 93.36%. In Figure 4.9, it can be seen that the NN learns effectively the highly nonlinear decision boundaries that separate each classification group in the complex plane since the predicted labels (right) are almost the same for all cases to the ground-truth (left). The most pronounced discrepancy is the misclassification of few 'good' instances as 'critical' eigenvalues (close to $\mathrm{Re}\{s\} = -0.2$).

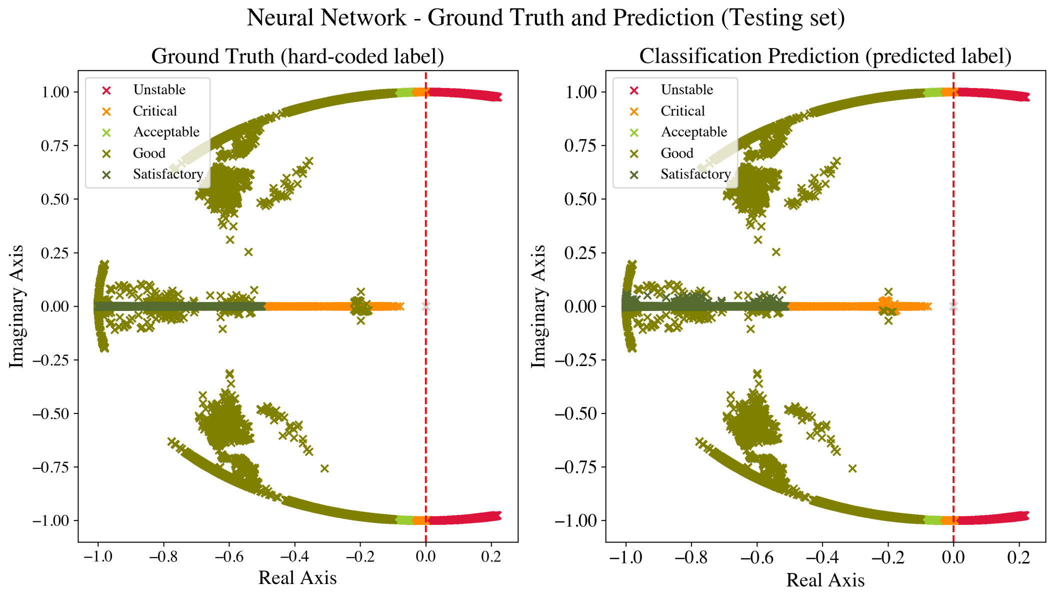


Figure 4.9: Ground truth and prediction results for the trained NN.

To quantify the performance and benchmark the different algorithms against each other, a simple numerical score was defined. Let $\mathbf{t}_{\text{ex}}$ be a vector containing the prediction time for every algorithm. $\mathbf{a}_{\text{train}}$ and $\mathbf{a}_{\text{test}}$, $\mathbf{p}_{\text{train}}$ and $\mathbf{p}_{\text{test}}$, and $\mathbf{r}_{\text{train}}$ and $\mathbf{r}_{\text{train}}$ are vectors containing the information about accuracy, precision and recall for the training and testing set, respectively. Then, the score for the ith algorithm is given by:

$$\begin{aligned} s^{[i]} = 0.4 &\left(\frac{\min\left(\mathbf{t}_{\text{ex}}\right)}{t_{\text{ex}}^{(i)}}\right) \\ &+ 0.2\left[\frac{1}{3}\left(\frac{\mathbf{a}_{\text{train}}}{\max\left(\mathbf{a}_{\text{train}}\right)} + \frac{\mathbf{p}_{\text{train}}}{\max\left(\mathbf{p}_{\text{train}}\right)} + \frac{\mathbf{r}_{\text{train}}}{\max\left(\mathbf{r}_{\text{train}}\right)}\right)\right] \\ &+ 0.4\left[\frac{1}{3}\left(\frac{\mathbf{a}_{\text{test}}}{\max\left(\mathbf{a}_{\text{test}}\right)} + \frac{\mathbf{p}_{\text{test}}}{\max\left(\mathbf{p}_{\text{test}}\right)} + \frac{\mathbf{r}_{\text{test}}}{\max\left(\mathbf{r}_{\text{test}}\right)}\right)\right] \end{aligned} \tag{4.8}$$

This score benefits the method with the minimum execution time which maximizes training and testing accuracy. For this reason, k-NN has a poor score despite its high accuracy, precision and recall performance. Testing has a higher weight than training on the final score since the algorithm is exposed to new instances, and therefore this number is a better indicator of generalization. The results for each method are presented in Table 4.2 where the accuracy, precision and recall are computed on the testing set.

Table 4.2: Performance metrics and score for ML/DL classifiers.

Method	t_{pred}	**Acc**	**Prec**	**Rcl**	**Score**
ζ classifier	5.477 s	100%	100%	100%	0.6032
LogReg	0.058 s	78.20%	64.09%	79.07%	0.7671
SoftmaxReg	0.054 s	78.10%	68.22%	83.02%	0.8067
Linear SVM	0.051 s	67.79%	41.61%	36.82%	0.6228
k-NN	33.027 s	**99.83%**	**99.68%**	**99.92%**	0.5997
Naïve Bayes	0.255 s	97.11%	86.42%	93.93%	0.6091
Decision Trees	0.057 s	98.93%	93.44%	96.19%	0.8933
NNs	**0.045 s**	98.53%	92.49%	95.90%	**0.9750**

4.5 Scalability

The proposed method to generate contingency scenarios can be scaled to deal with larger systems both in terms of buses (BUS) and number of states (STA) and variables (VAR). Note that for line openings, the maximum number of scenarios (SC) depends on the number of branches in the model (either transmission lines or transformers, written as BR in Table 4.3).

Systems with more than 30 branches (such as the Nordic 44) were constrained to have a maximum of five simultaneous contingencies (i.e., so that $N - 5$ is considered) to avoid sampling of unrealistic scenarios. Table 4.3 illustrates how the number of scenarios increases for different systems along with the estimated contingency pool generation time or execution time of the program (ET). This scalability benchmark was performed on a Ubuntu 18.04.5 LTS machine with a AMD Epyc 7601 32-core processor and 512 GB of RAM.

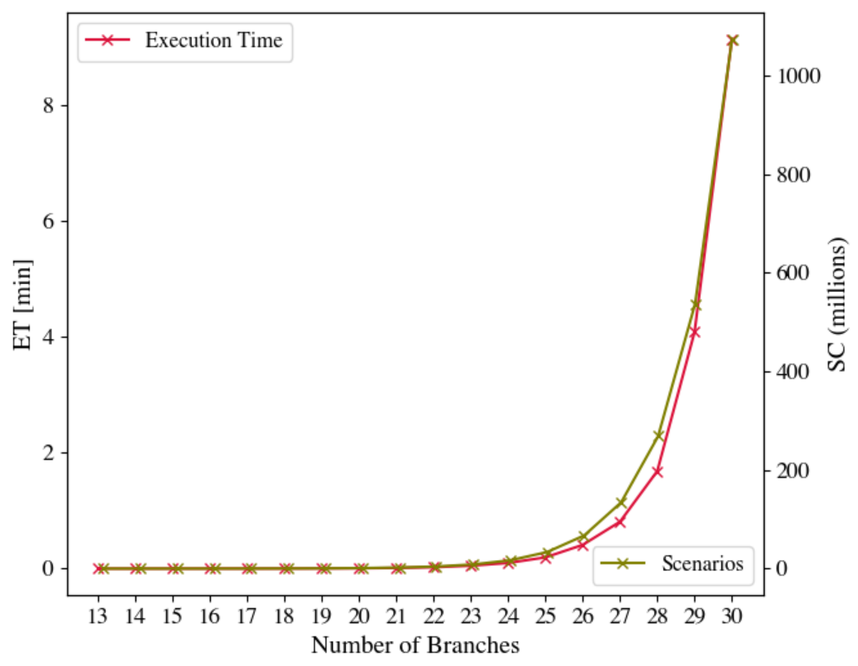
Table 4.3: Scalability of scenario generation for several systems.

System	BUS	STA	VAR	BR	SC	ET
IEEE 9	9	24	203	9	510	0.0348 s
Seven Bus	7	132	678	18	262,142	0.0781 s
IEEE 14	14	49	426	20	1,048,574	0.3038 s
N44	44	1294	6315	79	24,122,225	5.5966 s

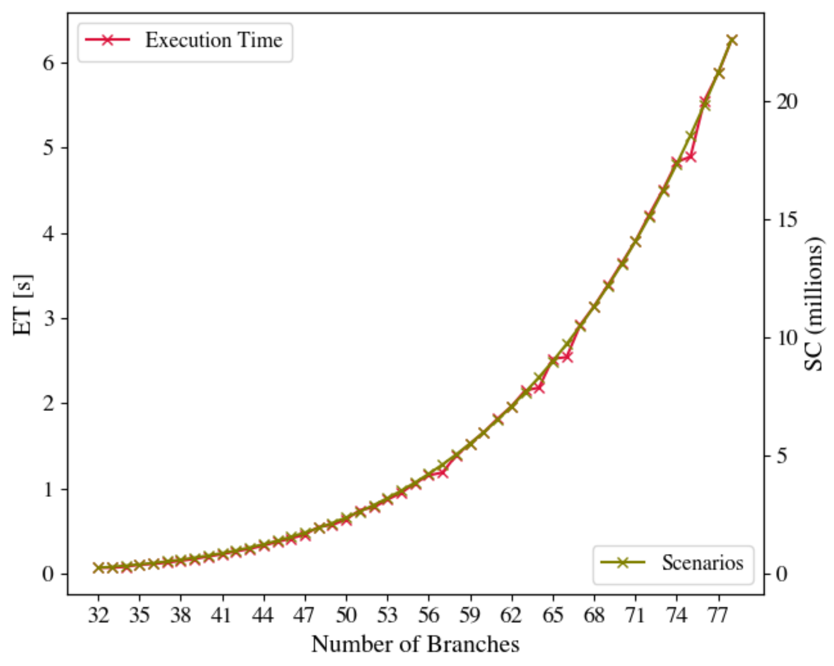
The effect of constraining the number of lines that can be simultaneously opened in a scenario not only enhances the practical significance of the method but also increases the computational efficiency when working with large systems for dynamical studies. In Figure 4.10a we see that the execution time grows exponentially as the number of branches increases requiring several minutes for a system with $\approx$ 30 branches. This number can represent relatively low interconnected grid models. Thus, we see that the method requires an adjustment for dealing with large-scale highly-interconnected systems.

By setting an upper bound on the number of maximum simultaneous contingencies, the execution time diminishes (from minutes to seconds) since the number of possible combinations is truncated. However, the number of scenarios is still significant: around 20 million for a system with $\approx$ 80 branches which can be obtained in less than 10 seconds. This suggests that working with larger grids would need to limit further the number of simultaneous trippings.

It must be stressed that the total amount of contingencies constitutes the sampling pool for the second stage of the proposed algorithm. The larger the pool is, the faster the method samples an arbitrary number of scenarios. It is clear that a larger scenario pool will ease the search for feasible contingency conditions.



(a) Without constraining the number of simultaneous outages.



(b) With an upper bound on the number of simultaneous line trippings.

Figure 4.10: Scalability of the contingency generation algorithm.

4.6 Conclusions

Through this chapter, we have seen how several ML methods can be employed to classify eigenvalues representing the behavior of a power system after the occurrence of a contingency. In particular, a decision tree and a NN have shown to be almost as accurate as a conventional classifier solution which computes the damping ratio for each eigenvalue while predicting faster. In this example, a considerable number of scenarios were studied by automating Modelica-based power system simulations thanks to the PDI. The use of the PDI enabled to integrate the development of the ML to the power system simulation. The trained NN showed classification performance above 95% for the testing data. This promising result highlights the potential of ML methods for deployment in real-time intelligent power system solutions.

CHAPTER 5
MODELICA GRID DATA: SOFTWARE TOOL FOR DATA GENERATION TO DEVELOP MACHINE LEARNING SOLUTIONS

5.1 Introduction

This chapter introduces `ModelicaGridData`: a data generation tool relying on massive phasor time-domain Modelica simulations employing the OpenIPSL. `ModelicaGridData` provides a pipeline to tackle the need for big data generation describing a broad range of operating conditions and potential contingencies experienced by a power system. This need is induced by the necessity to develop ML solutions, which depend on data-rich with power system dynamic behavior. Such solutions are required for the safe integration of renewable energies and the modernization of the aging infrastructure of the electric systems. Successful application examples include ML-based automatic oscillation detection, real-time small-signal stability assessment, and action recommender system developed using `ModelicaGridData`'s input.

`ModelicaGridData` implements algorithms to process different types of input data, perform steady state computations, run dynamic simulations and linear analysis routines, and label the data sets. It provides means for data scraping to use actual real-world load profiles from the publicly available information of a power system operator (see Chapter 3). The load profiles are used to compute different starting power flow conditions to use in the dynamic simulation of the models. The simulations are performed on several contingency scenarios to extend the generated cases beyond the normal operating conditions prevailing in most recorded power system measurements. The tool uses built-in functions within two different Modelica IDEs to generate small-signal stability labels for each scenario. As a sample application, labeling for small-signal stability assessment is automated, where labels

Portions of this chapter appear in S. A. Dorado-Rojas, F. Fachini, T. Bogodorova, G. Laera, M. de Castro Fernandes, and L. Vanfretti, "ModelicaGridData: Massive Power System Simulation Data Generation and Labeling Tool using Modelica and Python," submitted to the journal SoftwareX in June 2022 [9].

that correspond to the stability of the system are provided from linear analysis and nonlinear simulation time series analysis.

The tool has been developed entirely in Python 3 and is compatible with the Modelica IDEs, Dymola[1] and OM[2]. In addition, it has been designed to work with OpenIPSL models developed in versions 1.5.0 and 2.0.0. Moreover, it has been conceived as a cross-platform tool compatible with Windows and Linux operating systems.

5.2 Motivation and significance

Electrical grids are considered worldwide as critical infrastructure. Given its complexity and the limited opportunities to perform real-world experiments, model-based analysis has become an established approach to studying their behavior. In particular, phasor time-domain simulation models are designed to evaluate the grid response after the occurrence of a disturbance (such as a fault or a line outage), accounting for the effects of the system dynamics and the associated control systems. Dynamic simulation has become a ubiquitous tool in assessing power system stability, developing grid expansion plans and interconnection studies, performing root-cause analysis, and carrying out controller design and parameter re-tuning.

Several proprietary and commercial tools (e.g., PSS®E, PowerWorld) and alternative open-source software (e.g., Power System Analysis Toolbox (PSAT), Power System Toolbox (PST), Symbolic Power System Modeling and Numerical Analysis Library (ANDES)) are available for dynamic studies, with PSS®E being the most widely employed by utilities in the United States and used by 140 countries according to its developer, establishing itself as the *de facto* industry standard. In particular, in recent years, the OpenIPSL has emerged as an alternative to commercial tools, with promising potential in education and research [15]. OpenIPSL exploits the equation-based object-oriented paradigm of the Modelica language to represent phasor time-domain models. In contrast to other modeling alternatives for power systems, OpenIPSL does not require a discretization of differential equations for simulation; instead, the exact equation is used directly for modeling. Furthermore, in OpenIPSL 2.0.0, most dynamical components have been cross-validated, using several Modelica-compliant tools, against their PSS®E counterparts, yielding consistent results [5]. This capability

[1]See Dymola's website.
[2]See OM's website.

gives confidence to power industry specialists on the validity of results when using Modelica-compliant tools, thereby reducing users' resistance to change [54].

Given the fidelity of the results compared with PSS®E, OpenIPSL can be appropriately used as a powerful simulation tool to generate data representing dynamics in the phasor timescale. Data generation in power systems is highly relevant nowadays as an essential aspect for developing ML solutions, necessary for the safe accommodation of renewable resources and new technologies into the existing electrical grid infrastructure. Despite the proliferation of PMU, several barriers for utilizing their data exits. For example, non-disclosure agreements (NDAs) restrict the access of the public to the measurement data[3] [55]. Therefore, synthetic data generation approaches offer a vital source of synthetic measurements and system information for the "data hungry" ML-based development pipelines producing the ML solutions for the future power grid.

In power systems synthetic data generation solutions can be classified into two main groups: statistical and simulation-based methods. On the one hand, several solutions use available samples of actual PMU data (e.g., [56], [57]). The new data instances are generated by matching the statistical properties of the input data to a parametrized probability distribution. However, these approaches are constrained by the availability of PMU data, a vast amount of which is recorded during normal operating conditions due to the resiliency and robustness of power systems. Using these data to train ML model can lead to skewed distribution towards normal operation, where the instances related to abnormal operating conditions are not well represented (e.g., US power grid [58]).

An alternative to generating realistic data containing significant abnormal operating conditions is to use an existing simulation model to recreate different operating conditions. Special attention has been paid to developing synthetic models by learning the topological features of a network (i.e., the interconnection of buses and lines) [59], [60]. `ModelicaGridData` exploits the availability of an existing (synthetic) power system model to produce large-scale data. `ModelicaGridData` simulates a wide range of operating condition using a validated model of a power system. This approach has the main advantage of setting the grid condition for dynamic simulation to a realistic contingency scenario. The tool allows to simulate network conditions that are not likely to be seen in the real world,

[3]While it is possible to collect data for medium- to long-term timescales such as daily and hourly load and generation profiles (see, for instance, Open Power System data), data concerning transient power system dynamics is normally disclosed due to electricity market clauses.

albeit feasible, and where the system dynamics play a major role. To complement real system measurement with rarely occurred conditions, the developed software facilitates data generation describing abnormal operating conditions enriching the synthetic datasets with features of excited dynamics and, therefore, enable, potentially, to train ML models more accurately using complete data.

5.3 Software description

`ModelicaGridData` is designed to generate massive data using dynamic simulations of a Modelica power system model. Different scenarios are created by a) changing the starting steady state condition (e.g., power flow) of a power system and b) setting a scenario following an *ad-hoc* contingency scenario selection rule [8]. Contingency scenarios can also be generated using a graph theory-based technique using statistical characteristics of a real electrical grid [61]. After completing the simulations, the user can extract selected signals from the system simulations under all scenarios, getting "labels" for each simulated condition to perform small-signal stability assessment of the system.

`ModelicaGridData` allows the user to run the simulations on two different Modelica IDEs: Dymola (proprietary) and OM (open-source). Considerable efforts have been made to guarantee compatibility with OM, whose Python application program interface (API) is not extensively documented as Dymola's. Regardless of the selected tool, it has been found that performance for power system simulation is similar between the Dymola and OM [6] IDEs. Given the significant changes between OpenIPSL version 1.5.0 and 2.0.0, compatibility with standard settings of OM is guaranteed in `ModelicaGridData` for OpenIPSL version 1.5.0 only. For simulations using Dymola, OpenIPSL versions 1.5.0 and 2.0.0 are supported. The complete pipeline of the tool has been tested in Dymola 2021 and OM 1.16.2 on both Windows 10 and Linux (Ubuntu 20.04 LTS) operating systems.

5.3.1 Software Architecture

The program is structured into five modules fully integrated into the back-end (illustrated in Fig. 5.1), and an additional functionality whose use is shown through a Jupyter Notebook (see Section 5.3.2.6). The high-level idea of the application is as follows: using the New York Independent System Operator (NYISO) data (see Chapter 3), the profiles of the loads in a given system are varied using realistic profiles (`nyiso`). Then, multi-snapshot

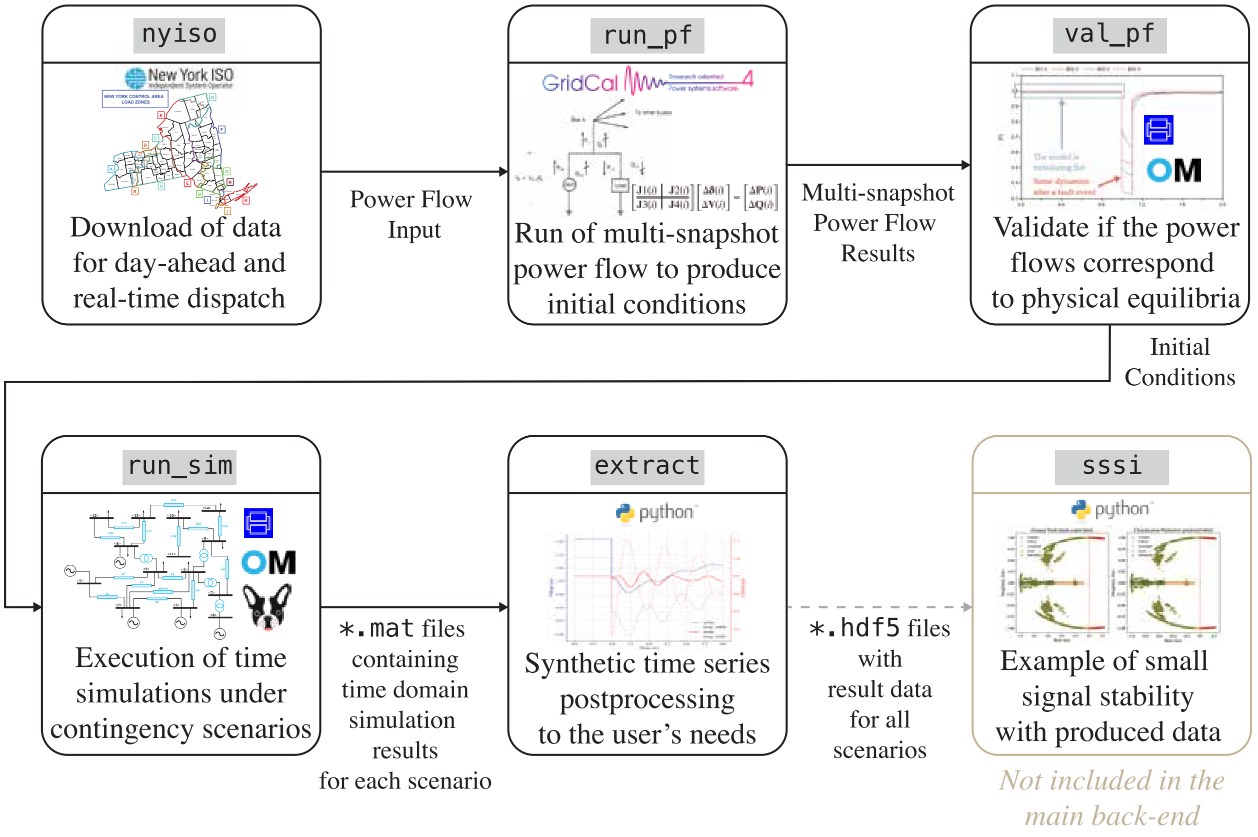

nyiso
Download of data for day-ahead and real-time dispatch
Power Flow Input
run_pf
Run of multi-snapshot power flow to produce initial conditions
Multi-snapshot Power Flow Results
val_pf
Validate if the power flows correspond to physical equilibria
Initial Conditions
run_sim
Execution of time simulations under contingency scenarios
*.mat files containing time domain simulation results for each scenario
extract
Synthetic time series postprocessing to the user's needs
*.hdf5 files with result data for all scenarios
sssi
Example of small signal stability with produced data
Not included in the main back-end

Figure 5.1: Architecture of `ModelicaGridData`.

power flows are computed (`run_pf`) and used to provide an initial guess from where the initial conditions of the dynamic model can be determined. The power flow data is loaded into the Modelica-based dynamic model and provided to a Modelica-solver (Dymola or OM), which solve the initialization problem (i.e. find the initial conditions for all dynamic states and perform the dynamic simulations. Because there is no guarantee that a power flow solution will lead to an equilibrium point of the dynamic model, each power flow result is validated to confirm whether the dynamic simulation with no events is initializing flat or not (`val_pf`) (see Chapter 3). As a result, only the physically meaningful power flows are kept. Then, the system under study is simulated under different initial conditions and contingency scenarios (`run_pf`) to generate synthetic data. The simulation results, stored in `*.mat` files, one per simulation, are then post-processed and organized into a single `*.hdf5` file (`export`) containing all the results for a specific simulation batch.

Each of the five modules that comprise the Python back-end is listed below:

1. Load data scrapping to generate power flow scenarios with realistic data (`nyiso`);

2. Power flow computation using the downloaded load profiles (`run_pf`);

3. Power flow validation (`val_pf`);

4. Massive time-domain simulation including analytic/symbolic linearization-based labels[4] (`run_sim`);

5. Time-series data extraction (`extract`).

The user can select a power system model among the available in the OpenIPSL library, or add a grid model of their own using the models of power system components in the OpenIPSL library and appending a corresponding PSS®E power flow model (`*.raw` file). The `*.raw` model for the corresponding system must be added in the `PSSE_Files` sub-directory inside the `./models` folder of `ModelicaGridData`. The power system models included as examples are the IEEE 14 bus system (`IEEE14`), the Single Machine Infinite Bus (`SMIB`), the IEEE 9 bus system (`IEEE9`), the Kundur Two Area system (`TwoAreas`), and the three-machine voltage regulator test system (`AVRI`)[5].

Once the power flow inputs are set, the steady-state computation is carried out by using the open-source Python-based library GridCal. GridCal is a powerful power system package for static computations. One of the important features of GridCal is that it provides native support to parse PSS®E data files (i.e., `*.raw` files for power system power flow models, while producing very similar power flow solutions with results up to the nonlinear algebraic equation solver's tolerance [7]. Then, the power flow solutions are validated using (`val_pf`), the validation checks if the dynamic model can be initialized and if a dynamic simulation results in a "flat" trajectory. If the validation is successful, the validated power flows are incorporated as attributes that define initial guess values used in the phasor-based Modelica model of the system selected by the user.

When the power flow conditions used as the starting point for the time domain simulation are set, the next step to generate the data is to run different contingency scenarios under several power flow conditions. The tool is designed to perform the simulations using Modelica power system model using either the Dymola or OM Modelica tools via their

[4]The Modelica language profits from the equation-based modeling including the analytically evaluation of the derivatives in the system state equations. These results are used for computing analytic linearization to get a linear model explicitly from the derivatives. The module `run_sim` employs such linearization at initial and final conditions of the model simulation.

[5]The authors encourage the user to develop their own models following the structure in the examples.

corresponding Python APIs. The results *for each simulation* of the system's model are stored in a single `*.mat` file, which is inconvenient for data extraction of massive simulation batches (i.e., 20,000 simulations). Finally, the `*.mat` files are "unified" into a single `*.hdf5` file.

Besides the five modules, the tool is provided with a Python implementation of well-known power system operating state classification algorithms that are employed for the data labeling in case of small-signal stability assessment [49].

The five modules included in the main back-end of the tool can be executed in a command prompt by running a driver file called `main.py`. Detailed instructions to set a virtual environment with the required dependencies are given in the GitHub repository accompanying this chapter. The format of the command to run `main.py` is as follows:

```
main(function, [,kwargs])
```

where `function` corresponds to the name of the module to be executed (see Fig. 5.1), and `kwargs` are the corresponding keyword arguments of the module (see Section 5.4).

It must be underlined that we prepared two VirtualBox virtual machines (Windows 10 and Ubuntu 20.04 LTS) that can be used for off-the-shelf testing of the tool. However, the greatest advantage of `ModelicaGridData` is an ability to use the available computational resources, e.g., exploiting multi-core capabilities for simulation and batch execution. A step-by-step guideline on how to set up the tool in a standalone computer for each operating system is also provided in the GitHub repository.

5.3.2 Software Functionalities

In this section, we will present the major functionalities of the software, including a simple example with a corresponding command to run each module. We emphasize that the documentation for each module and all the functions is available under `./docs/doc_functions.md/` in the associated repository.

5.3.2.1 Load Data Scrapping: `nyiso`

This module, identified as `nyiso`, scraps the publicly-available data from the NYISO website [62] and organizes it per load regions of the New York grid. The New York grid has 11 load regions, identified by a particular keyword, namely, Capital (`CAPITL`), Central

(`CENTRL`), Dunwoodie (`DUNWOOD`), Genesee (`GENESE`), Hudson Valley (`HUD VL`), Long Island (`LONGIL`), Milwood (`MILLWD`), New York City (`N.Y.C.`), North (`North`), and West (`WEST`). The data correspond to the day-ahead and the real-time scheduling used for establishing the dispatch in the NY grid.

The data scrapping algorithm fetches the `*.csv` files published in the NYISO web page [62] and creates a single spreadsheet for each of the load regions for a given date. It classifies the registers into measurements and forecasts. The measurements correspond to the real consumption for each region at each time stamp. The forecasts contain the short-term predictions made for any given date. Two different instances named "best" and "worst" forecasts are produced. The "best" forecast contains the latest forecast produced for a given date. Likewise, the "worst" forecast has the information regarding the first predictions for a given date (typically published four days in advance in NYISO). By working with the real measurements and the forecasts, the user is able to analyze the impact of forecasting error on the grid's dynamic behavior. The `nyiso` module is called as follows:

```
python main.py nyiso [--date] [--path]
```

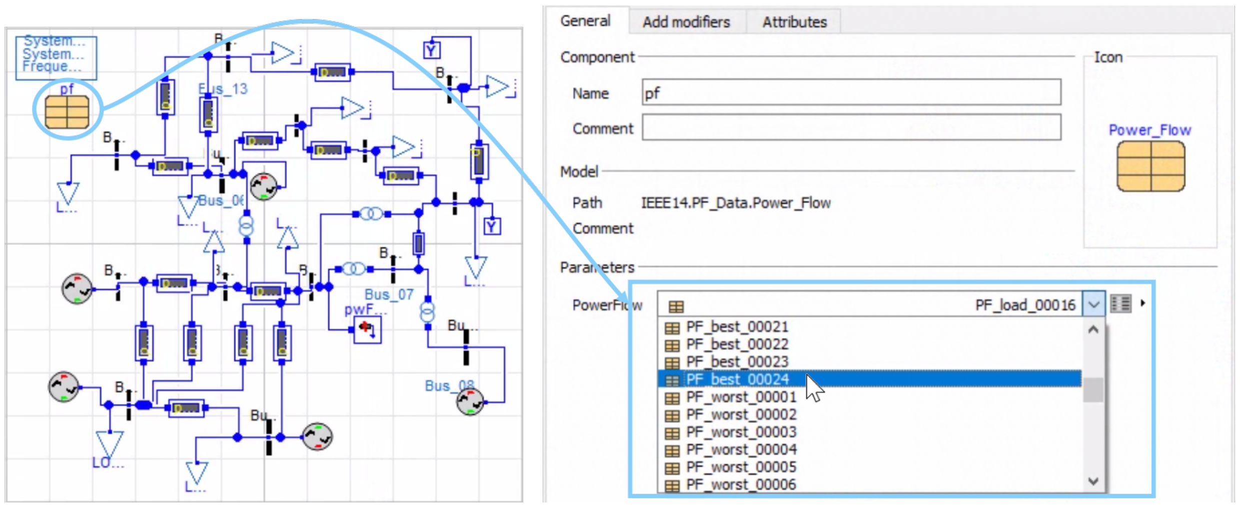
where `--date` and `--path` are two arguments that specify the starting date to extract data and the path where the data will be stored, respectively (see Section 5.4 for a specific example).

5.3.2.2 Power Flow Computation: `run_pf`

The module `run_pf` in Fig. 5.1 takes the data scrapped from the NYISO website (`https://www.nyiso.com/`) and uses it as *patterns* to vary the loads in a power system model. For each snap shot of the load pattern, GridCal computes a power flow, that if validated, it will later be used to start the dynamic simulations of the model. In other words, the loading levels $\alpha_{\ell,\text{area}}$ are varied according to the trend as seen in the corresponding NYISO profile:

$$\alpha_{\ell,\text{area}} = p_{\text{area}} / \max(p_{\text{area}}), \tag{5.1}$$

where $0 \leq \alpha_{\ell,\text{area}} \leq 1$ and p_{area} is the corresponding load level for an area from the NYISO data. Then, the load level used for the power flow computation is



Name pf

Comment

Path IEEE14.PF_Data.Power_Flow

Comment

PowerFlow PF_load_00016

PF_best_00021
PF_best_00022
PF_best_00023
PF_best_00024
PF_worst_00001
PF_worst_00002
PF_worst_00003
PF_worst_00004
PF_worst_00005
PF_worst_00006

Figure 5.2: Power flow selection in the graphical interface of Dymola. Each power flow result is generated after the execution of the `run_pf` module. The power flow selection is automatic in the back-end.

$$p_{\ell,\text{area}} = \alpha_{\text{area}} p_{\text{bc}} \tag{5.2}$$

where p_{bc} is the base case load for the system under study at each zone.

The power flows are generated by varying some (or all) of the loads in a system. The pairing between load regions and loads in the model is done through a random mapping (i.e., the area pattern used for varying the level of each load in the model is assigned randomly). The power flow grid model is loaded from the corresponding PSS®E file (`*.raw`), and the power flow solution is found using GridCal's solvers [63]. The power flow results are parsed into a Modelica `Records` file, following a data structure for OpenIPSL models as explained in [7] (see Chapter 3). Such object-oriented data management approach allows to replace the power flow results by modifying only *one* attribute at a time, that is directly connected to the *several* variables that are related to the power flow computation result (see Fig. 5.2, however, note that such change does not need to be applied graphically/manually, but is instead applied automatically by `ModelicaGridData`).

The module `run_pf` is invoked using the command:

```
python main.py run_pf [--model] [--version] [--window] [--date]
[--loads] [--delete] [-seed]
```

A detailed discussion of each parameter in the command is provided in the documentation shared on GitHub.

5.3.2.3 Power Flow Validation: `val_pf`

The `val_pf` module evaluates whether a power flow result is correct, in terms of compliance with the grid model constraints, when initializing the model at steady-state. This is done using a short-duration dynamic simulation (i.e., spanning 5 s or less) and evaluating the deviation between the initial and final values of the dynamic states. Such short-spanned, event-free simulations do not represent a significant computational burden in the Modelica-compliant IDEs if the `dassl` solver is used [6] (see Chapter 2). Then, the `val_pf` module relies on automated time-domain simulations to carry out the validation procedure. The call of this module is done as follows:

```
python main.py val_pf [--model] [--version] [--tool]
[--proc] [--cores] [--pc]
```

`--pc` indicates whether a virtual machine (`--pc vm`) or a physical computer (`--pc pc`, default) is used. Then, the simulation parameters in the file `val_parameters_pc.yaml` or `val_parameters_vm.yaml` are loaded. The user can modify the simulation settings in either of these two files if desired. Moreover, `--proc` and `--cores` relate to the fact that the `val_pf` function implements a combined parallel-sequential[6] approach to dynamic simulations. In other words, the power flow results can be divided between several processes (`--proc`) to speed up computation. Each process will use a number of `--cores` physical cores to parallelize the internal computations, i.e., it parallelizes a simulation execution among many cores. The reader is referred to the GitHub documentation for gaining more insight into the arguments of the module's main function. An example is provided in Section 5.4.

5.3.2.4 Massive Dynamic Simulation: `run_sim`

Once the validated power flow solutions for a given model are available (see Section 5.3.2.3), the user can dispatch massive phasor time-domain simulations using the `run_sim` module (Fig. 5.1). The function starts a set of dynamic simulations with distinct operating

[6]A parallel-sequential approach is understood as the simultaneous execution of multiple processes each of which uses a single physical core of the host machine. Each process is assigned with a mini batch of the complete simulation scenarios.

conditions (power flows) under several contingency scenarios. The function and its arguments to perform massive simulations are as follows:

```
python main.py run_sim [--model] [--version] [--tool]
[--proc] [--cores] [--pc] [--n_pf] [--n_sc] [--n_sim]
```

The user selects the power system model (`--model`) and the OpenIPSL version (`--version`) to use in the execution process. Furthermore, to run the simulations the Modelica tool can also be chosen between Dymola (`--tool dymola`, default) and OM (`--tool om`). All the execution flags to speed up the simulations using the DAE solvers on both tools have been implemented according to [17], [18]. However, OM simulations are only supported with OpenIPSL version 1.5.0.

Regardless of the environment that has been selected for the simulations, the user can also specify the number of power flows (`--n_pf`), contingency scenarios (`--n_sc`), and the maximum number of dynamic simulations (`--n_sim`) to be executed. Furthermore, the code allows for parallelized simulations via assigning the arguments `--proc` and `--cores` similar to the `val_pf` module (see Section 5.3.2.3).

5.3.2.5 Data Extraction: `extract`

The `extract` module (see Fig. 5.1) is designed to work in interactive mode with the user being requested inputs on which signals to extract. First, the user is suggested to select a component within the model to extract dynamical signals from. The implemented options of choice that are related to the basic OpenIPSL models are bus, line, and generator. However, this can be easily modified and expanded by the user to include a choice of other devices such as STATCOMs [64], advanced converters [65], and renewable sources [66].

Each simulated scenario generates three `*.mat` files containing 1) the time series data, 2) the initial-condition linearization output, and 3) the final-condition linearization results. With this data structure results are well organized, however, dealing with a case where, for instance, 20,000 scenarios are available, would be cumbersome. The main goal of the `extract` module is to help the end-user organize the output of `run_sim` into a single file such that the data can be distributed and read easily, while at the same time, giving well organized results for other post-processing purposes or applications that can run without the need of user interaction from the files themselves. The command to call this function is as follows:

```
python main.py extract [--model] [--tool] [--version]
[--mu] [--sigma] [--pc]
```

where `--model` is a name of the system model employed for simulations, `--tool` is the Modelica-compatible IDE used with `run_sim` module (more details are in Section 5.3.2.4), and `--version` is the OpenIPSL version on which the model has been built. Note that these parameters are shared with both `val_pf` and `run_sim` modules.

To make the resultant time series resemble real measurements better, the user has the option to introduce additive Gaussian noise to the results of interest. The mean and the standard deviation of the normally-distributed noise are controlled by the arguments `--mu` and `--sigma`. More information about the default values is found in the GitHub documentation of the tool.

5.3.2.6 Small-Signal Stability Synthetic Data Labeling: `sssi`

The labeling module of the tool aims to assess the small-signal stability condition of the power system. For this purpose, the classical Prony's method is implemented in Python as the `prony` function. This classical method requires data preprocessing, therefore, the `filtering_data` function is added to the labeling module. The result of Prony method is processed by `sssi` function that generates stability indices for small-signal stability assessment. Thus, the module has three functions `filtering_data, prony, sssi` [49], [67].

The pre-processing function `filtering_data` defines a process that extracts the data after a contingency from the synthetic measurement data. This is needed because the non-stationarity of the studied signal during a contingency may lead to inconsistent results from Prony's method, which assumes a linear model. Thus, the function removes a portion of the input data that is related to the contingency process, thus providing only the oscillation content that corresponds largely to the linear system's response. The signal is then detrended in order to get the oscillatory behaviour of the data set. To perform detrending, an average detrending is used. Also, to filter out noise and the trend, another option is proposed that is based on the highest energy content: Intrinsic Mode Function (IMF).

The Prony function takes as input the detrended data set and estimates the eigenvalues of the signal after passing through a frequency screening process that excludes frequencies outside an inter-area mode range.

The `sssi` function uses the detrended and frequency filtered data and generates three stability indices based on the damping ratio of the eigenvalues from the Prony function: Single Mode Index (SMI), All-Mode Index (AMI), and Global Mode Index (GMI) [49].

5.4 Illustrative Examples

An illustrative example concerns data generation using the IEEE 14 bus system. The reader can replicate each of the steps described in this section in either of the two virtual machines provided with the code. Note that running the code in the Dymola IDE requires a license (successful execution with the trial version included in the virtual machines is not guaranteed). The reader is referred to the YouTube playlist accompanying this chapter for a hands-on example demonstration [68].

First, the `nyiso` module (see Section 5.3.2.1) is executed. This is done by running the following command, while located in the parent directory of `ModelicaGridData`:

```
python main.py nyiso --year 2020 --path data
```

The tool will download all the available data from the NYISO website [62], starting from January 1st, 2020, up to the day before today, as indicated by the optional argument `--year`. The data is saved under a sub-folder relative to the current directory named `data`. By default, the tool fetches the data from the year in course (e.g., starting January 1st 2022 if the code is run in 2022). Therefore, care should be taken with the value of `--year` passed when calling this module. The data from the NYISO website are available starting from 2001. Moreover, NYISO updates their current registers at 12:00 PM EST each day. So, if the most recent information is required, the scrapping module should be run after this hour.

Next, the scrapped data is used to compute several power flows with GridCal. The following command is employed to run power flows varying three loads in IEEE 14 bus model (built in OpenIPSL version 2.0.0) at a time:

```
python main.py run_pf --model IEEE14 --version 2.0.0 --loads 3
```

When the power flow results are available, the next step is to carry out the validation process via the `val_pf` module. The command to run this step is shown below. The validation will be executed according to the simulation settings specified in `val_parameters_pc.yaml`. By default, it spans a five-second simulation with no events. With the settings below, two processes running each instance of Dymola are created. Each process will use one physical core to complete the validation of the results.

```
python main.py val_pf --tool dymola --model IEEE14
--proc 2 --cores 1
```

Once the power flows that do not result to initialization in an equilibrium point are removed (if any), the user can proceed with running `run_sim` module (see Section 5.3.2.4). An example command is presented below:

```
python main.py run_sim --tool dymola --model IEEE14
    --proc 4 --cores 2 --n_pf 2 --n_sc 10 --n_sim 16
```

The program will create four processes, each using two cores. Two power flow results will be used for each contingency scenario. So, in parallel the maximum possible scenarios that could be run is $2 \times 10 = 20$. However, the user specifies a maximum of 16 simulations by the argument `--n_sim`. So, instead of the 20 potential scenarios that potentially could be run, only 16 will be completed among the four processes created, for a total of 4 per process.

Finally, we employ the `extract` function to collect all simulation results for a simulation batch. This function will organize all the signals of interest from the available `*.mat` files in the working directory into a single `*.hdf5` file. The example of call to the `extract` module is:

```
python main.py extract --tool dymola --model IEEE14
```

The `extract` module requires the input of the user to navigate through the `*.mat` files and get the desired results. Then, there will be several user prompts throughout the execution of the function. A complete example is available in the GitHub repository of the tool. The plot of the signals obtained from the `*.hdf5` file [69] is shown in Fig. 5.3.

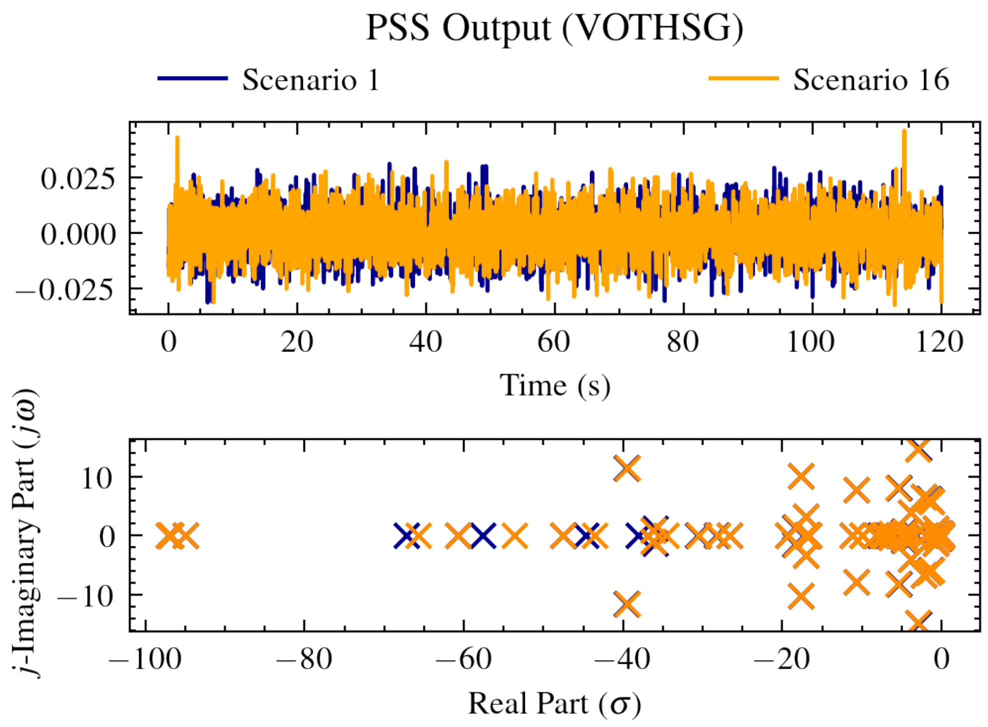


Figure 5.3: Visualization of the extracted data using the `extract` module.

5.5 Impact

The proposed tool represents a pipeline for massive labeled power system simulation data generation using Modelica model. In addition, the Modelica models developed using OpenIPSL library [15] are initialized with power flow solutions using records that was not available before. Given that data generation and pre-processing represent a significant resource-consuming stage for developing ML solutions[7], the tool introduces a degree of automation that spares much of the cost and effort in creating such applications. Potential applications include (but are not limited to) the development of edge modules for oscillation detection [12] and time series-based small-signal stability algorithms [10] (see Chapter 6).

To the authors' best knowledge, other model-based simulation frameworks employ dynamic simulation as a grid emulation method and not as a data generation mechanism[8], so, our approach represents the first milestone toward high-fidelity simulation-based data generation for the production of machine learning data-driven algorithms to assist power

[7]Former IBM's senior vice president Arvind Krishna (currently CEO) claimed approximately 80% of the work in developing a novel ML solution is related to data collection and preparation [70].

[8]See, for instance, GridSTAGE

system engineers in the operation of 21st-century electric grids.

5.6 Conclusions

`ModelicaGridData` is a data generation pipeline capable of producing synthetic power system time series measurements upon phasor time-domain simulation of power systems using Modelica under several contingency scenarios. The tool exploits the automation capabilities of Python to download and create power flow results that work as initial conditions for a dynamic simulation using Modelica-based IDEs. The user can start simulation along multiple processes using two different Modelica IDEs in either Windows or Linux operating system. After the simulations are completed, the generated data can be extracted in a compact convenient form at the user's will by an interactive module that returns the synthetic measurements for all simulated scenarios in a single `*.hdf5` file.

In addition, this tool allows to make the best use of the available offline resources (e.g., deployed computing servers in utility companies or research institutions) to speed up the data generation processes. Through a simple example, we have demonstrated the usability of the application and its user-friendliness. Since the tool relies on the open-source power system library OpenIPSL, experienced users can tailor their models to their particular needs, including expansion of the models with other components within larger system models.

CHAPTER 6
TIME SERIES-BASED SMALL-SIGNAL STABILITY ASSESSMENT USING DEEP LEARNING

6.1 Introduction

The standard approach of small-signal stability analysis (SSA) [50] is to quantify the effects of the small disturbances, such as a line trip, by analyzing the stability properties of the linearized power system model. The obtained model is usually presented in the form of a state-space representation $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$. Then, linear system analysis is applied to assess the small-signal stability condition by evaluating the system's eigenvalues, thereby obtaining the damping ratio of the dominant modes [50]. When the model is not available, a measurement-based mode identification technique, such as Prony [71], is applied. Prony requires recording of several swing oscillations to get acceptable accuracy in mode identification.

Despite the usefulness of the linear analysis and measurement-based mode estimation techniques to evaluate SSA, their implementation for a real-time analysis poses several challenges. Such drawbacks arise from the complexity of the required computations for a large-scale system (e.g., maintaining a validated model) or the measurement data requirements to obtain acceptable accuracy (e.g., filtering).

DL is a family of ML algorithms that are based on artificial NNs with feature learning capability [72]. In other words, these algorithms allow extracting essential elements of the data that define an output that has to be learned by the DL algorithm.

Some ML algorithms have been recently applied in power systems to enhance and evaluate small-signal stability. For example, the work of [46] performs coordinated tuning of PSS parameters using heuristic optimization algorithms. Likewise, in [47], a cuckoo search is employed to find optimal PSS parameters that guarantee small-signal stability. Moreover, in [48], the parameters of a unified power flow controller are tuned via an NN whose weights

Portions of this chapter appear in S. A. Dorado-Rojas, T. Bogodorova, and L. Vanfretti, "Time Series-Based Small-Signal Stability Assessment using Deep Learning," presented at the 2021 North American Power Symposium [10].

are optimized using the Levenberg-Marquardt algorithm. Despite some efforts to study SSA with conventional ML techniques, there is no research on the small-signal stability assessment using deep learning methods to the authors' best knowledge.

DL algorithms can be trained using collected measurements and data generated by performing offline power system simulations (see Chapter 5). The trained algorithm has to be deployed for inference on real-time data to provide fast identification of the state in which the system operates. However, in this new approach, the main challenges are: (i) to choose the effective DL algorithm, (ii) to select the best data and appropriate amount of data to train the algorithm with sufficient accuracy while (iii) ensuring an acceptable real-time performance. This chapter aims to provide insight into these challenges.

This chapter's main contribution is a proposal to use and a comparison of the training results of state-of-the-art DL algorithms for small-signal stability assessment using time series input power system data (either synthetic (simulated) or the measurements from a real power system). Special attention is paid to the case studies on the measurement selection (voltage or current measurements) and data preparation for the algorithms' training. For this purpose, case studies are performed for voltage and current measurements with 1% Gaussian noise (added in the preprocessing) and without noise of a simulated model to assess small-signal stability. The performance of studied architectures is discussed and compared using the performance evaluation metrics such as accuracy, precision and recall. The architectures are: a Multi-Layer Perceptron, a fully CNN, a time convolutional neural network (t-CNN), an inception network, and a multi-channel deep convolutional neural network (MCDCNN). Hyperparameters of the algorithms, such as the effective number of epochs, are also reported. Furthermore, the trained algorithms' prediction/classification time per data sample is presented to analyze which model is most suitable for a real-time application.

This chapter expands the work in [8] (see Chapter 4) by considering a time-series data as input directly, rather than the set of eigenvalues describing a particular operating condition. In [8], we used the output from several dynamic simulations to compute the eigenvalues characterizing a particular condition. Eigenvalues had to be further preprocessed before classifying the operational condition of the system using the decision boundaries learned during training. In this work, the pre-processing task of identifying the system eigenvalues is a feature learning task performed by the particular layers of the proposed NN architectures.

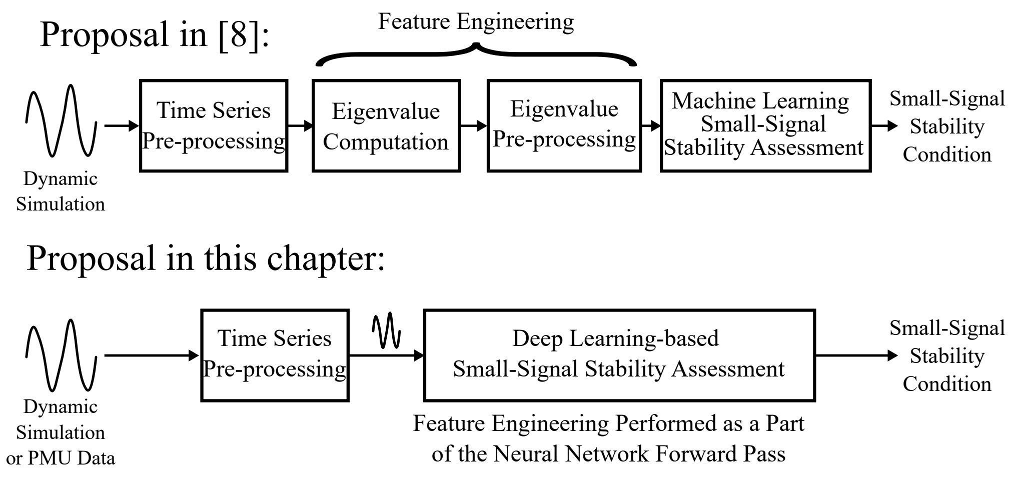

Proposal in [8]:
Feature Engineering
Dynamic Simulation
Time Series Pre-processing
Eigenvalue Computation
Eigenvalue Pre-processing
Machine Learning Small-Signal Stability Assessment
Small-Signal Stability Condition
Proposal in this chapter:
Dynamic Simulation or PMU Data
Time Series Pre-processing
Deep Learning-based Small-Signal Stability Assessment
Small-Signal Stability Condition
Feature Engineering Performed as a Part of the Neural Network Forward Pass

Figure 6.1: Approaches to SSA by conventional ML and DL.

Since unstable operating conditions are rare, they are created via model-based simulation. To this end, we follow the automated phasor time-domain simulation approach that is based on realistic selection of a set of contingencies for power systems described in [61] to produce training data. The procedure in [8] follows a different contingency generation algorithm.

In contrast to our previous work, using time series data as input expands the applicability of the methodology and enables the potential inclusion of PMU data for training. Both approaches require a time series pre-processing stage, but the approach of this paper does not require any eigenvalue computation. We emphasize that the feature engineering task (i.e., computing eigenvalues from time-series data) is carried out in the forward-pass of the NN. This is not possible with the classic ML methods applied in [8] where the pre-processing had to be done beforehand. The difference between both approaches is underlined in Figure 6.1.

This chapter is structured as follows: Section 6.2 presents an overview of SSA from time-series data and a description of the studied DL architectures. Section 6.3 describes the data generation and pre-processing stages. In Section 6.4, we show the results of each case study and discuss NN performance for SSA. Finally, Section 6.5 elaborates on common challenges of the proposed deep learning application and Section 6.6 concludes the work.

6.2 Small-Signal Stability and Deep Learning

6.2.1 Time Series-Based Small-Signal Stability Assessment

The proposed approach (Fig. 6.2) to perform SSA consists of an offline and an online step. The former includes data preparation (time series preprocessing and labeling) and the deep NN training; the latter refers to exploitation with the trained NN on non-labeled data to carry out SSA.

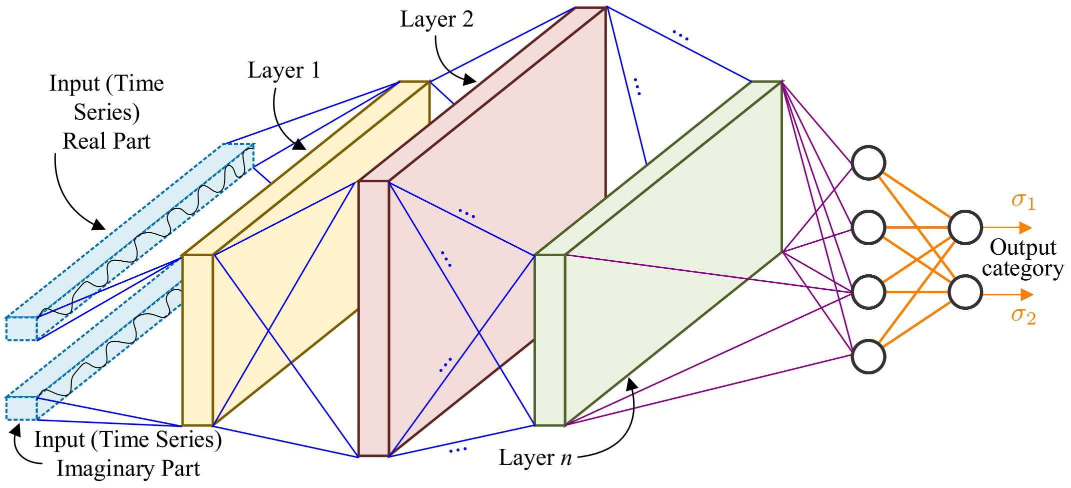

Input (Time Series) Real Part
Layer 1
Layer 2
σ_1
Output category
σ_2
Input (Time Series) Imaginary Part
Layer n

Figure 6.2: Convolutional neural network structure for time series classification.

6.2.2 Deep Learning for Time-Series Classification

Since this is a supervised learning problem, we need to provide a set of examples for the DL algorithms to learn how to categorize the data. These examples are the time series traces with labels. The labels indicate whether a trace corresponds to either of the classification categories. Labeling is performed using symbolic linearization of the grid model to get a system matrix. Then, the dominant eigenvalue is used to compute a damping ratio that defines the corresponding label (stable/unstable) for a given scenario. Thus, in our experiments a training instance is composed of a phasor time-domain profile of a generator bus voltage or line current (see Fig. 6.3) and a binary label calculated offline via model linearization.

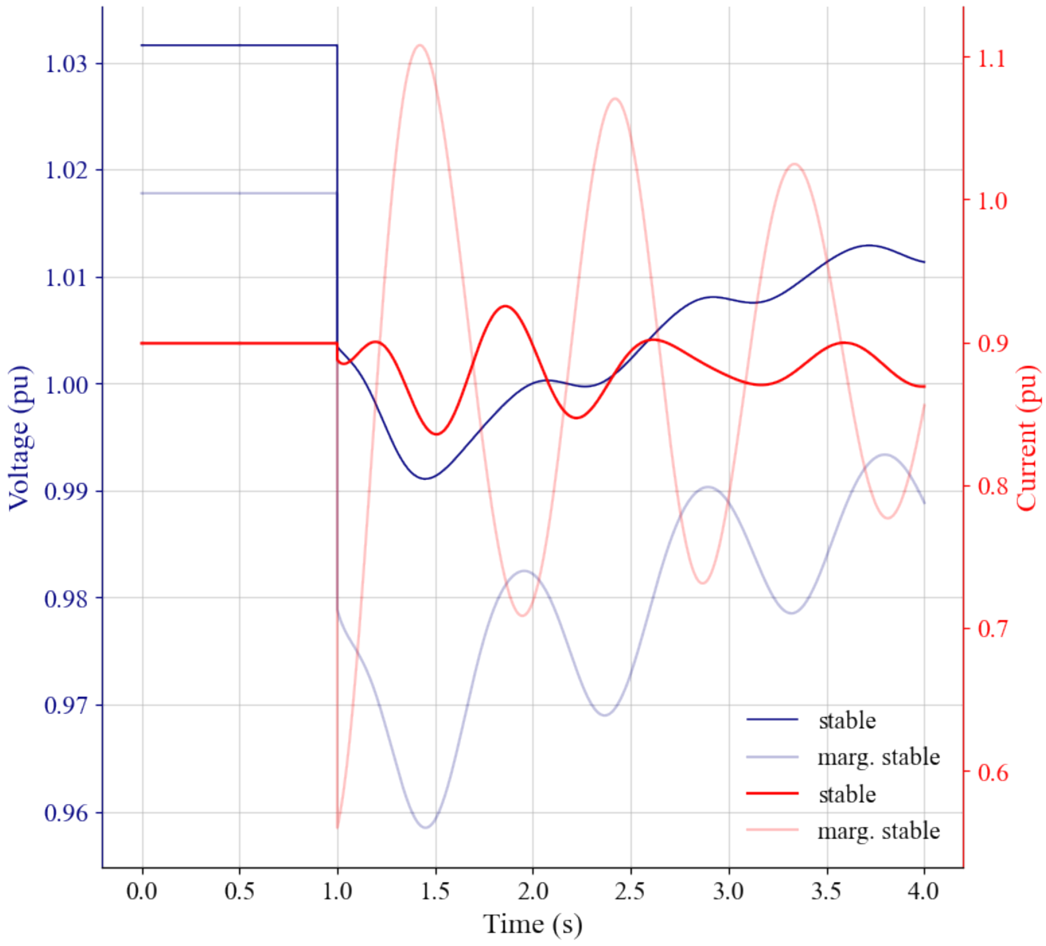


Figure 6.3: An example of synthetic current and voltage magnitude measurements for stable and marginally stable scenarios (the disturbance occurs at $t = 1.0$ s).

During training, the NN learns to identify the characteristic patterns of each category by updating its parameter values iteratively. After training, a deployed NN can classify/define the label for the operating condition by feeding voltage or current data directly. In this way, the NN performs feature engineering to internally find a representation of the time-series data that will allow it to discriminate the input directly. So, system identification is left as a task for the NN.

DL stands from a NN architecture with (deep) numerous interconnected layers of reduced number of neurons in each layer without loss in performance. In general, DL algorithms consist of three main components: a NN architecture, a cost/loss function, and an optimization method. The architecture is related to the NN structure, which includes the number of neurons in each layer, the number of layers, and the type of layers. The cost/loss function represents the criterion based on which the NN improves its performance while learning from the given examples. Finally, the optimization method provides the mechanism to update the NN parameters, such as weights in the functions that represent the neurons.

The goal of training process is to find the optimal set of weights w_i using the training set's information so that the NN's output will maximize a performance criterion or minimize a loss function.

The fundamental building block of an NN is an artificial neuron, a black-box model inspired by biological synapses. Given an input $\mathbf{x} \in \mathbb{R}^{N\times 1}$ with N features $\mathbf{x} = [x_1\ x_2\ \ldots\ x_N]^T$, a neuron performs a weighted sum of the components of $\mathbf{x}$ and returns an output y after composing the weighted sum with a simple function, mostly nonlinear, referred to as activation function σ. The output is given by:

$$y = \sigma\left(\sum_{i=1}^{N} w_i x_i\right).$$

The output y is fed to another neuron in a deeper layer. In this way, an NN represents a continuous composition of weighted nonlinear functions.

A case of particular interest is when the input is a time-series (i.e., a sequence showing temporal dependency through its components). In this case, the vector $\mathbf{x} \in \mathbb{R}^N = [x_1\ x_2\ x_3\ \ldots x_{N-1}\ x_N]^T$ is composed by N measurements/observations. The problem of classifying a given time-series $\mathbf{x}$ into a pre-specified category is known as time series classification (TSC). In this section, we briefly describe the CNN architectures for TSC employed in this paper. All hyperparameters for training were taken from [73].

6.2.2.1 Multi-layer Perceptron

A multi-layer perceptron (MLP) [74], or a deep feedforward network, computes an output by the weighted sum of every input signal component, followed by a pointwise nonlinear activation. When deployed for TSC, an MLP's performance may downgrade due to its architectural properties. The temporal interdependence in the input is not captured because all components are weighted individually before being passed to each layer. Despite this drawback, an MLP [74] is used as a base case to compare with more sophisticated NNs for TSC since it represents a lower performance bound.

The baseline MLP has four fully-connected layers. The first input layer corresponds to the size of the input data. The three hidden layers have 500 neurons per each layer with a ReLU activation function for each neuron. To avoid overfitting, Dropout regularization layers are added prior to each hidden layer correspondingly. The dropout rates are 0.1, 0.2 and 0.2, respectively, Finally, the output layer is a dense layer of size that is equal to number of classes with a softmax activation function and a prior dropout rate equal to 0.3.

6.2.2.2 Convolutional Neural Networks

CNNs are architectures inspired by the structure of the human eye with great success in the field of image processing. The convolution aims to filter the input signal's content to learn the most relevant features effectively. The training objective is to learn the filters that help extract the most from the filtered signal. This network type is suitable for data showing a grid-like structure, such as images (2D grid of pixels) and univariate time-series (1D grid of samples).

CNNs use convolution as a mapping operator in at least one of their layers. The convolution[1] of $\mathbf{x}_N$ with a filter $\mathbf{w} * N$ is

$$(\mathbf{x}_N \ * \ \mathbf{w}_N)\,[n] = \left(\sum_{i=1}^{N} x_i w_{i+n} \right). \tag{6.1}$$

Table 6.1: DL model characteristics for SSA.

Model	Parameters		Optimizer	Layers		
	Trainable (T)	Non-trainable (NT)		Layer Number	Type (activation)	Hyperparameters
`fcn`	265,986	1,024	Adam	8 (4T + 4NT)	3x conv1d (ReLU) 1x dense (softmax)	$n_{\text{filters}} = (128, 256, 218)$ kernel size = (8, 5, 3) padding = same
`inception`	422,850	2,048	Adam	64 (17T + 47NT)	16x conv1D (ReLU) 1x dense (softmax)	$n_{\text{filters}} = 32$ kernel size = 41 padding = same
`mcdcnn`	1,443,526	0	SGD	15 (8T + 7NT)	7x conv1D (ReLU) 1x dense (softmax)	$n_{\text{filters}} = 8$ kernel size = 5 padding = same
cnn	1880	0	Adam	7 (3T + 4NT)	2x conv1D (sigmoid) 1x dense (softmax)	$n_{\text{filters}} = (6, 12)$ kernel size = (7, 7) padding = valid
mlp	1,007,502	0	Adadelta	8 (4T+ 4NT)	3x dense (ReLU) 1x dense (softmax)	$n_{\text{neurons}} = 500$

A CNN has several advantages: sparse interactions, parameter sharing, equivariant

[1]In the context of DL, convolution corresponds to cross-correlation, not to the convolution operator in the context of linear time-invariant systems.

representations, and ability to work with inputs of variable size. Sparse interactions mean that the output does *not* interact with every input unit, allowing memory reduction (fewer parameters) and efficiency boost (fewer operations). Likewise, parameter sharing indicates that the same parameter is used on several layers, which improves computational efficiency. The equivariance property for dealing with time series data means that when different features appear in the input, and a particular event is time-shifted, the same signal will appear in the output, shifted equally [72]. This means that if the input changes, the output changes in the same way.

A typical layer of a CNN includes three stages: *convolution*, *detector*, *pooling*. In the *convolution* stage, the layer performs convolutions on the input to get a set of outputs that run through a nonlinear activation function (e.g., ReLU) in the *detector* step. Finally, a *pooling* function replaces the output at a certain location with a summary statistic of the nearby outputs (e.g., max pooling returns the maximum value of the particular neighborhood). Thus, the output is invariant to noise.

In this work, four CNN architectures –fully convolutional neural networks (FCNs), inception, MCDCNNs, and t-CNNs are studied. The general structure of the CNNs is shown in Fig. 6.2. The CNN is composed of n convolutional layers (layers $1-n$ in Fig. 6.2) that followed by a fully-connected layer (purple in Fig. 6.2) and a softmax activation in the output layer (orange in Fig. 6.2). Let $\mathbf{z}$ be the vector of values at the last NN layer. A softmax activation σ for the binary case (i.e., $\mathbf{z} \in \mathbb{R}^2$) is computed and the category predicted as follows):

$$y = \operatorname{argmax}\left[\sigma\left(\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}\right)\right] = \operatorname{argmax}\begin{bmatrix} \dfrac{e^{z_1}}{e^{z_1}+e^{z_2}} \\ \dfrac{e^{z_2}}{e^{z_1}+e^{z_2}} \end{bmatrix} \tag{6.2}$$

A FCN (`fcn` in Table 6.1) has three convolutional layers followed by a batch normalization stage and a ReLU activation function each. The training is performed to minimize a cross-entropy loss function. The third convolutional layer's output is averaged over time dimension (global average pooling) before being fed into an output softmax layer.

A t-CNN (`cnn` in Table 6.1) [75] has two convolutional layers followed by softmax activation and average pooling operation each. The loss function is a mean squared error.

An inception network (`inception` in Table 6.1) [76] consists of 6 inception blocks –concatenated outputs of 4 convolution layers and one max pooling connected in parallel-followed with batch normalization operation and activation function, a global average pooling layer, and a fully-connected output layer. Thus, it takes a previous input and passes it to the parallel convolutional layers concatenating the outputs together with the output of max pooling operation over the input. So, a bigger variety of filters can be chosen in each layer.

An MCDCNN (`mcdcnn` in Table 6.1) [77] consists of typical CNN layers where the convolutions are performed in parallel on each dimension of the time series data. Each two convolutional layers have 8 output filters of length 5 with a ReLU activation function, followed by a max-pooling operation. The convolutional layers' output is flattened before entering a fully-connected layer with a ReLU activation function. Finally, the output layer is fully-connected of the size that corresponds to number of classes with a softmax activation function (see Eq. (6.2)).

6.3 Case Study

6.3.1 Data Generation

We have generated trajectories using the IEEE 9 bus system (24 state variables; 203 algebraic variables) initialized for a vast array of operating conditions (i.e. power flows) and subjected to realistic contingencies that are generated using the algorithm in [61]. Once the contingency scenario is applied, a dynamic simulation is carried out for 4 s to generate trajectories (see Fig. 6.3). A total of 1805 simulations were generated, from where $n_{\text{training}} = 1083$, $n_{\text{testing}} = 602$, and $n_{\text{validation}} = 120$.

6.3.1.1 Data Preparation

The voltage signals at generator buses contain system dynamics that, in practice, can be measured by PMUs. Thus, the real and imaginary parts of voltage at some generator buses and selected line current signals are used to construct training and testing datasets. Normalization is not required since all waveforms are per-unitized.

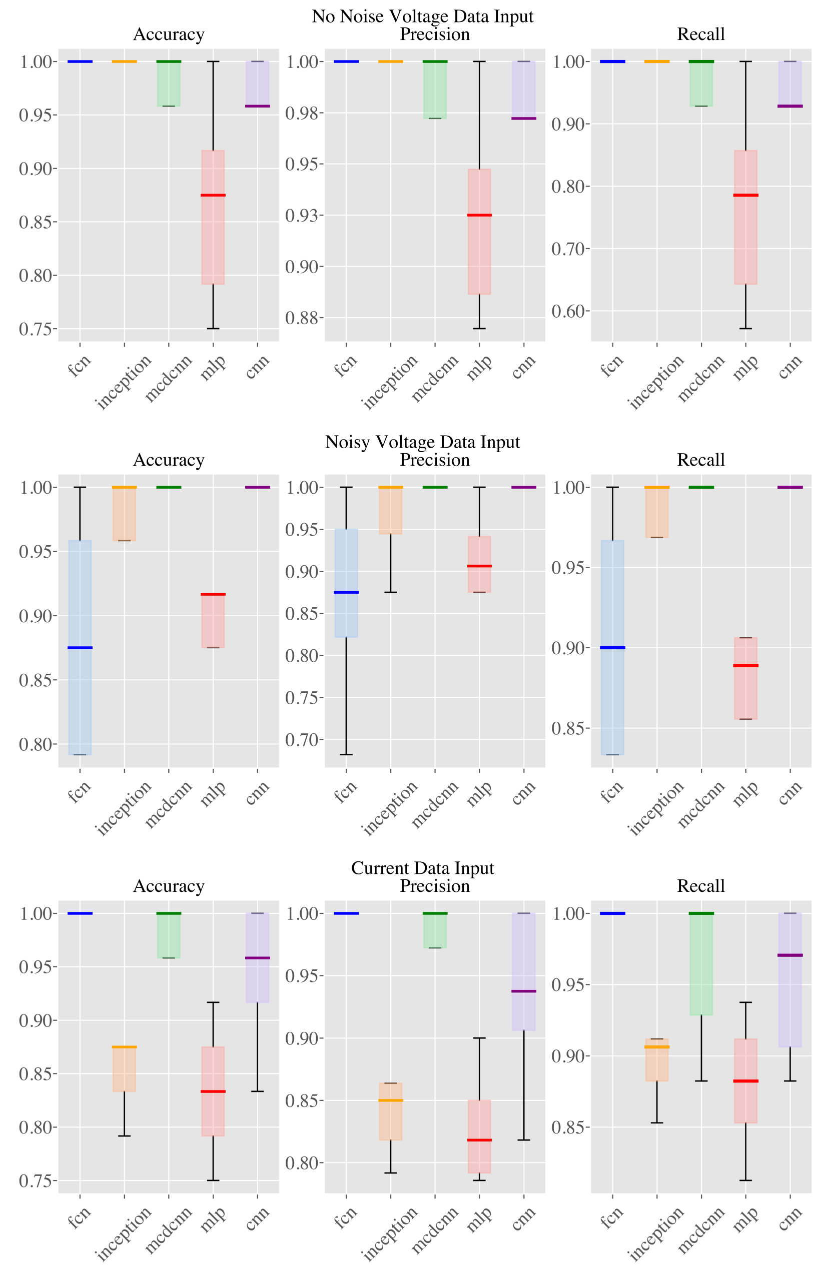


Figure 6.4: Training results for 5-fold cross-validation (full length data set).

Also, 1% Gaussian noise, typical for PMU measurements, is added to the voltage signals. In summary, three datasets that include pairs of real and imaginary parts of each signal (voltage, noisy voltage, and current) were generated.

The labeling of the datasets for training of the CNNs is performed using the classic SSA. Small-signal stability is analyzed either by linearizing a nonlinear grid model or identifying the excited dynamics from measurements (e.g., PMU data).

For ease of interpretation, we limit the NN task (Fig. 6.2) to the binary (two classes) problem of stable (when damping ratio of the system $\zeta > 5\%$) and marginally stable ($0\% < \zeta < 5\%$) classification.

To train and validate the models on the more complex learning problem, the resulting data sets are further preprocessed. Thus, to expedite SSA, the measurement length has been reduced to 75% of the original simulation time. The resulting length corresponds to a measurement window of 3 s. This would require less information to be passed to the NN. Likewise, the number of training instances is reduced by a factor of 3 ($n_{\text{training}} \approx 300$) to account for data scarcity.

For all experiments, the input tensors' shape is $(n_{\text{scenarios}}, T, 2)$, where T is the number of points in the time-series. The last element of the tuple, '2', indicates that real and imaginary parts are input data features.

6.4 Results

Results for the experiments with full and reduced length time series, with all and fewer instances, are presented in Figs. 6.4 and 6.5. Conventional splitting results are shown in Fig. 6.6.

6.4.1 Analysis

The number of samples for the presented case studies could be considered small when applying DL. Thus, k-fold cross-validation ($k = 5$) was performed to validate the performance of each DL model.

In Fig. 6.4, we observe that the CNN architectures with the tuned hyperparameters (`fcn` and `cnn`) can achieve 100% performance on the data generated for the experiment. This does *not* mean that the NN will be fully accurate when deployed but rather that the performance will be very high. Thus, in our setup, CNNs successfully detect the oscillation patterns allowing for distinguishing a stable operating condition from a marginally stable one.

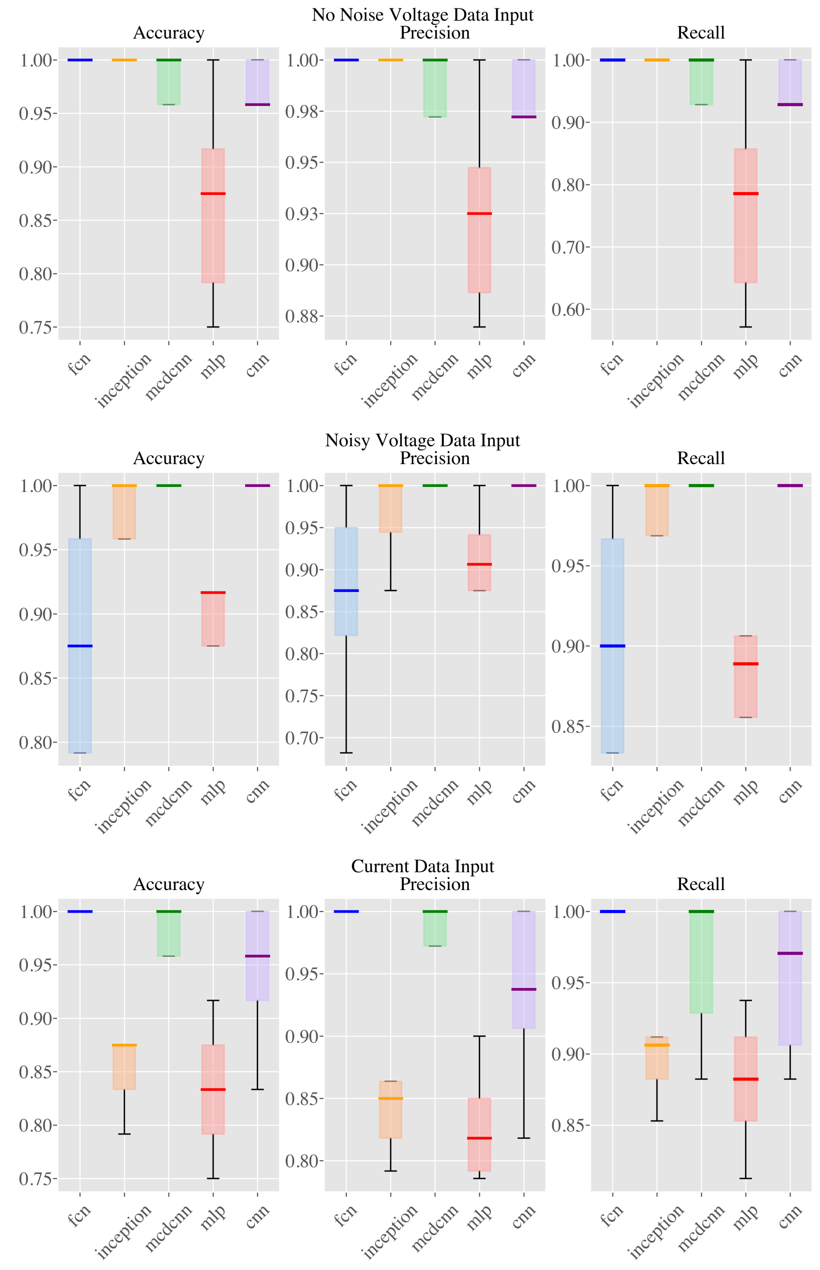


Figure 6.5: Training results for 5-fold cross-validation (reduced length data set).

Fig. 6.5 shows that performance is not significantly downgraded after reducing the time series length. Some architectures can relate the oscillating behavior with a category better than in the previous case.

Nevertheless, we observe that assessing the system's condition from current measurements could be more challenging than for voltage inputs. This is due to presence of more prominent oscillations in voltage measurements in comparison to the current traces from the training data.

Purely convolutional architectures (`fcn`, `mcdcnn`, and `cnn`) show metrics beyond 99% regardless of noise for both voltage and current inputs. In fact, the performance of all models is above 99% for noiseless voltage. However, the inception model lost its classification capability when exposed to noisy measurements (accuracy: 0.6761; precision: 0.3380; recall: 0.5000). For current data inputs (Fig. 6.6), the performance of the MLP degrades which is expected due to the limitations of this architecture for identifying patterns in time series data.

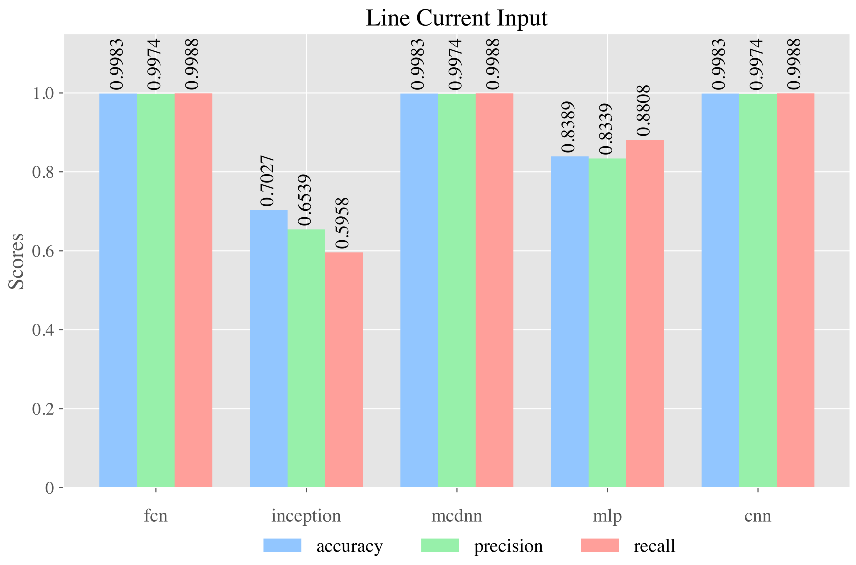


Figure 6.6: Prediction results for line current input (reduced training data set).

In the reviewed cases, the oscillatory pattern is easily learned by the convolutional layers since it contains the excited dynamics of the small system.

For the presence of more complex dynamics in the large systems, achieving high performance would require finer hyperparameter tuning such as bigger number of neurons and larger data sets for learning, which would be more computationally expensive. Despite this, we conclude that DL methods are powerful to learn classifying patterns that are typical for SSA.

6.4.2 Computational Performance Analysis

To measure the computational performance of the NN models in production, the prediction time per sample and the number of epochs to train the best model (see Table

6.2) were computed. The prediction time per sample indicates how fast the NN computes a prediction of the system stability given a time series input. The number of epochs is a relative measure of the training effort required to optimally tune the model. For all architectures, prediction time is in the order of milliseconds which favors the deployment in real-time pipelines.

MLP is the fastest architecture in terms of prediction, but also the one requiring a larger number of epochs to achieve its best performance. Moreover, there is a tradeoff between production performance and training effort for all models. The MCDCNN (Table 6.2) is optimal in both senses.

Table 6.2: Prediction time and number of epochs for best model.

Model	**Voltage**		**Noisy Voltage**		**Current**	
	Pred. Time	**Eph Best**	**Pred. Time**	**Eph Best**	**Pred. Time**	**Eph Best**
`fcn`	1.20 ms	58	1.21 ms	209	1.21 ms	133
`inception`	2.50 ms	**8**	2.43 ms	**11**	2.48 ms	**8**
`mcdcnn`	0.477 ms	59	0.478 ms	59	0.491 ms	45
`mlp`	**0.234 ms**	2226	**0.225 ms**	1760	**0.235 ms**	211
`cnn`	2.75 ms	248	0.282 ms	235	0.261 ms	165

6.5 Discussion

The presented training pipeline of classical state-of-the-art deep learning architectures is very common in the computer science community. However, when solving specific field engineering tasks such as small-signal stability assessment, several issues arise and must be addressed. The first issue is which data to use as an input to train the models and process these data. In this case, when PMUs are widespread in the power grid, we chose to use voltage or current data, either noisy or noiseless, to validate the selected NN architectures' ability to catch the patterns that define the system's state (stable or marginally stable). The second issue is to elicit the data enriched with such distinctive patterns, meaning to find the measured values after a contingency or create the big enough dataset of such synthetic measurements to achieve a good performance of the models. This challenge has been addressed in [61] whose algorithm has been adopted in this work to generate the data.

Another issue that is left open is a transfer learning possibility that can be applied to the best performance model. Transfer learning is characterized as a possibility to use the trained model for measurements collected from other power systems without significant degradation of the model performance. This point requires special attention and is left as future work. The next issue connected with the NN architecture's performance is how much data is enough to get an acceptable performance of the trained model. In this work, we gradually enlarged the amount of data until at least one model of the compared deep learning algorithms showed a good performance. But generally speaking, the larger the model, the more extensive dataset is required for training. In particular, if the power system is characterized by the significant number of eigenvalues that define the system's state, more combinations of the excited modes can be present after a contingency is applied. Therefore, more data is needed to train a deep neural network.

Eventually, the issue of the computational performance is significant if the trained model is applied in any control center. In Table 6.2 the prediction time per data sample is presented for each studied model. We consider that the trained model is used in a plug-and-play manner. This means that the presented time is enough to perform the small-signal assessment if the input data chunk is available. But one has to be aware that this chunk is of the length of 3 seconds. This is the time series length to capture enough patterns to classify the state of the system. Comparing the trained models to the classic Prony method performance, the deep learning models give a significant speedup in performance since the Prony method is effective only after the oscillations caused by contingency are fade out [71], [78]. Thus, for the Prony method to achieve a good performance, the needed time series length is of approximately 10 seconds in comparison to the 3 seconds that have been used in the presented case studies.

6.6 Conclusions

This work proposed a novel methodology for time series-based power system small-signal stability assessment using deep learning. The models are tested using the labeled current and voltage measurements generated using massive dynamic simulations after realistic contingency scenarios were applied. The accuracy, precision, and recall show a good performance of the trained models with the selected hyperparameters and for the generated data. Prediction time and performance show the potential of NNs to be deployed in real-time control center tools for operator decision support.

CHAPTER 7
LOW-COST HARDWARE PLATFORM FOR TESTING MACHINE LEARNING-BASED EDGE POWER GRID OSCILLATION DETECTORS

7.1 Introduction

Power system oscillations can be roughly categorized into free and forced [79]. Free oscillations occur permanently in the system around a stable equilibrium point and are naturally damped out by the system. On the other hand, forced oscillations, such as interarea modes, emerge when a power system is perturbed by external disturbances that excite its modes' natural frequencies [80]. A forced oscillation may cause incipient instabilities or severe equipment damage by inducing negative impacts on the power system. In extreme cases, it may even result in system breakup, power outages, and equipment damage if not detected at the right time. Consequently, it is vital to develop a method for detecting and locating forced oscillations *on time* to reduce their negative impact [81].

A myriad of methods for detection and mitigation has been proposed to address forced oscillations in electrical grids. In [82], two non-parametric methods are presented to estimate an oscillating mode. The authors emphasize the importance of monitoring power system modes in real-time and propose a technique to determine the existence and persistence of forced oscillations. Likewise, in [83] a Phasor Measurement Unit (PMU)-based real-time sub-synchronous oscillation detection pipeline is discussed. The experiment results are quantified via simple metrics to assess the performance of the involved hardware and software systems [84].

Such techniques rely mostly upon signal processing stages to ensure proper and timely event distinction. However, the computational complexity of such methods is usually

Portions of this chapter appear in S. A. Dorado-Rojas, S. Xu, L. Vanfretti, G. Olvera, M. I. I. Ayachi, and S. Ahmed, "Low-Cost Hardware Platform for Testing ML-Based Edge Power Grid Oscillation Detectors," presented at the 2022 10th Workshop on Modelling and Simulation of Cyber-Physical Energy Systems (MSCPES) [11].

significant. The online performance of such techniques has been recently assessed. The detection method in [85], for example, takes 1.7 s to process and label a forced oscillation. Similarly, the detection time in [86] is around 350 ms. These two samples of algorithmic time execution unveil the question of whether signal processing-based solutions are also feasible for real-time deployment. In particular, it is unclear if edge devices, such as those in future information exchange schemes such as the Internet of Things (IoT) [87], would have the capabilities to execute efficiently such heavy signal processing workflows.

Machine Learning (ML) emerges as an alternative data-driven paradigm to develop solutions because ML models are computationally efficient and nowadays simple to create. Beyond forced oscillation detection, ML has proven successful for other problems such as stability assessment [8], [10] and dynamic contingency management [88]. However, in power systems, a caveat is that measurement data describe mostly normal operating conditions. Several authors have started generating data via computer-based offline (e.g., [8]), and real-time simulations (e.g., [89]).

Either by conventional or by ML methods, the development of detection algorithms capable of real-time inference requires algorithmic efficiency *and* a suitable testing platform to generate or stream the oscillation data in real-time. A typical hardware platform for real-time experimentation is Hardware-in-the-Loop (HIL) simulation, in which the user can replicate the conditions of a particular engineering system with high accuracy. HIL has gained popularity not only in power systems but also in other domains (e.g., see [90] for an example of autonomous vehicles). In the context of oscillation detection, a HIL testbed has been used to validate the accuracy and feasibility of a fast PMU-based proposal [91].

While real-time simulation represents the most accurate way to produce training data and stream measurements during validation of real-time algorithms for oscillation detection, such simulators are not easily accessible because of their high price tag. Then, there is a need for a low-cost experimental platform to synthesize training datasets and validate detection algorithms in real-time. This chapter aims to bridge this gap.

We introduce a low-cost platform (see Fig. 7.1) for end-to-end validation of an ML solution using a low-voltage signal generator, namely an Analog Discovery 2. Such boards are commonly used in a first electronics class in most universities. Synthetic signals are generated thanks to the WaveForms Software Development Kit (SDK) programmatically. The waveform is streamed using an Analog-to-Digital Converter (ADC) to an NVIDIA Jetson

TX2 device. Such specialized hardware counts with a Graphics Processing Unit (GPU), and it is capable of executing ML models in real-time. A trained Convolutional Neural Network (CNN) model is downloaded in the Jetson board and used for real-time inference: given a user-defined waveform, the CNN predicts whether a forced oscillation is occurring or not based on the available information window. A discussion of the training process of the CNN is beyond the scope of the chapter. The reader is referred to [12] for more insight in this regard.

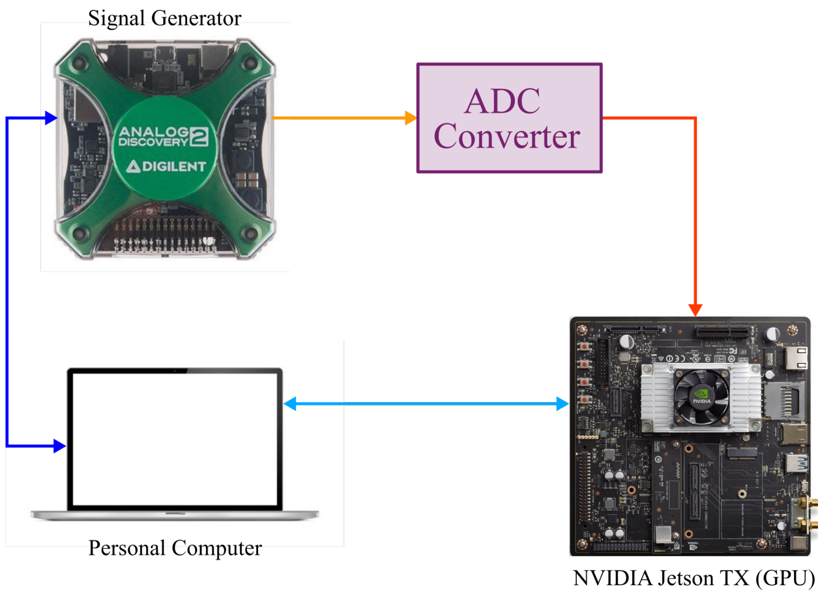


Figure 7.1: Proposed low-cost test platform.

In summary, the main contributions of this chapter are as follows:

1. we introduce a low-cost test framework that can be applied in the design phase for an ML-based oscillation detection algorithm before a full real-time simulator-based HIL test;

2. we develop a methodology for real-time signal emulation controlling an Analog Discovery 2 board using the WaveForms SDK;

3. we compare the inference performance when two different ADC are deployed;

4. finally, we present a way to automate the experiments using socket Application Programming Interface (API) in Python when the experiment process is time-consuming and laborious.

The reminder of this chapter is structured as follows: Section 7.2 introduces the mechanism to generate signals using the Analog Discover 2 board. Section 7.3 presents the two different ADC methods for benchmark. Experiment automation is described in Section 7.4. Results are discussed in Section 7.5. Lastly, Section 7.6 concludes the work.

7.2 Real-Time Signal Emulation

It is necessary to generate a synthetic signal to emulate a sub-synchronous oscillation, similar to those observed in PMU measurements [83], so that we can assess the performance of a CNN in real-time. Such a task is possible with the Analog Discovery 2 board thanks to the WaveForms SDK. The WaveForms SDK is a public API available in programming languages such as Python and C++. It allows users to interact with the Analog Discovery 2 board and automate testing via simple applications. The scope of this section is to describe at a high level how a signal with different patterns was generated using the SDK. Then, we briefly mentioned how noise is added to the generated signal to emulate measurement and process randomness.

7.2.1 Characteristics of Forced Oscillation Waveforms

In a steady-state, a power system dynamical state $\mathbf{x}$ is said to operate at a stable equilibrium condition $\mathbf{x}_{\text{eq}}$ when $\dot{\mathbf{x}}|_{\mathbf{x}=\mathbf{x}_{\text{eq}}} = f(\mathbf{x}_{\text{eq}}) = \mathbf{0}$. The continuous stochastic nature of loads makes the state $\mathbf{x}$ oscillate around $\mathbf{x}_{\text{eq}}$. In practice, observables of power grids (e.g. measurements such as current and voltage phasors) show "small-amplitude oscillations" around an equilibrium condition. A forced oscillation is characterized by an abrupt change in the amplitude of a signal in a power grid lasting a specific time. If the oscillation is stable, the power system should return to equilibrium after the event fades out. However, stable forced oscillations could lead to cascading events that can lead to a massive event such as a blackout. Detecting forced oscillations is critical to take remedial actions that guarantee a reliable operation of any electrical system [92].

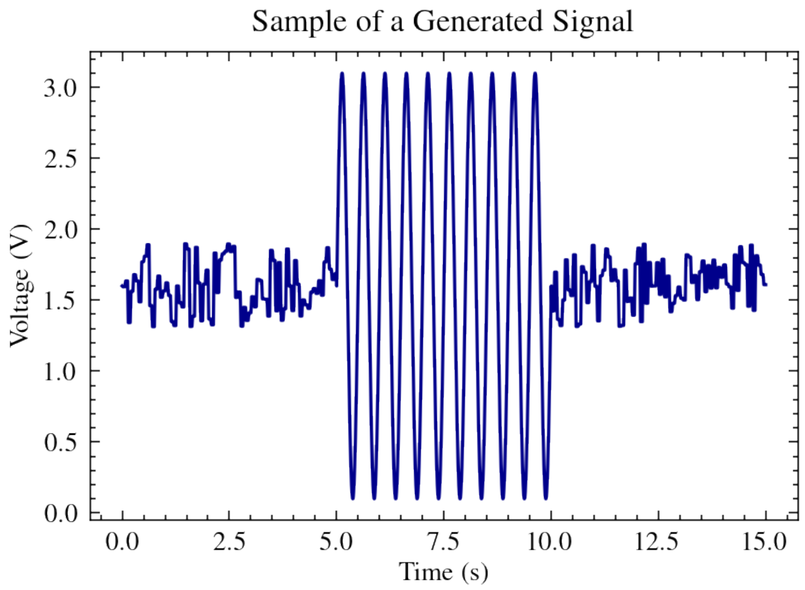


Figure 7.2: Example waveform describing a forced oscillation. The signal was produced with the Analog Discovery 2 board.

Assume an event leading to a forced oscillation has occurred in the grid. Before the contingency, measurements will show a stationary behavior characterized by the excursions of the system state around $\mathbf{x}_{\text{eq}}$. During the event, the amplitude of the signal will change significantly. The system will return to equilibrium if the proper remedial actions are taken, e.g. ramp-down of a wind farm's power output [83]. An example waveform characterizing a conceptual forced oscillation event is shown in Fig. 7.2.

7.2.2 Generation of Signals using the WaveForms SDK

Based on the previous discussion, the most simple signal that characterizes a forced oscillation consists of three parts: an oscillating behavior around a steady-state value; a high-amplitude oscillating waveform during the event; a final noisy-like segment that describes the return to the equilibrium condition. Such signal is generated straightforwardly using the WaveForms Python API. The API allows the user to modify each part of the signal by varying frequency, amplitude, and offset parameters for different shapes. The particular form in Fig. 2 was generated by a random signal (sampled from a uniform distribution) with a pre-specified 1.5 V offset. After 5 s, the output of the signal generator is changed to a sinusoidal signal to mimic a forced oscillation event lasting for 5 s. Lastly, the output is switched again to a noisy signal. Several test signals with different characteristics can be easily generated by amplitude and/or frequency sweeps by following this simple workflow.

To increase the "complexity" of the generated waveform, noise is superimposed also on the oscillation part as shown in Fig. 7.3.

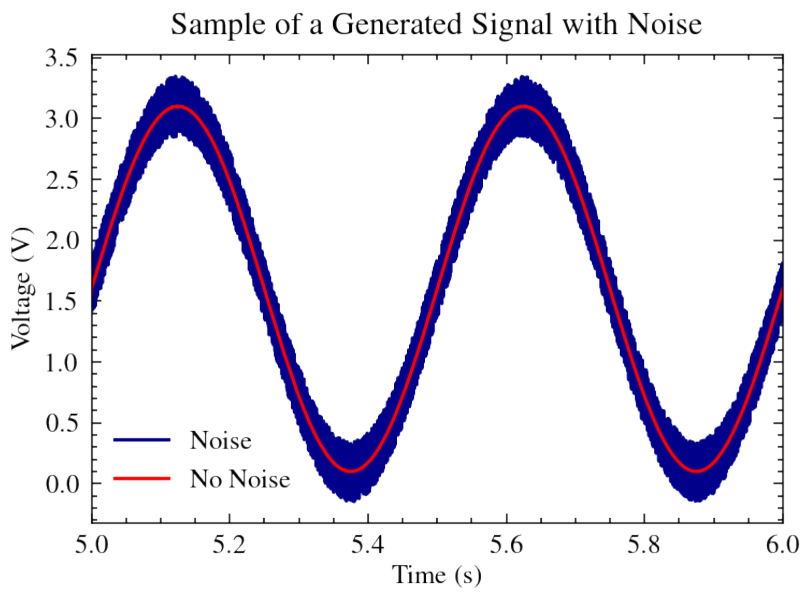


Figure 7.3: Example waveform with noise superimposed on the forced oscillation sinusoid.

7.3 Data Acquisition Methods

After generating a signal waveform for testing real-time inference in an ML-based oscillation detector, the next step is to send the signal to the NVIDIA Jetson TX2. This requires a data acquisition or conversion stage. The core of this step is the ADC. ADCs are a mature technology whose main advantages are fast conversion and low cost, so their use in the context of the proposed platform is justified.

Following the ADC conversion, the data is transmitted to the Jetson via I^2C. Therefore, in this section we explain how the ADC and the I^2C communication stages are implemented. For benchmarking purposes, two different ADCs were tested, namely an 8-bit ADC (PCF8591) and a 12-bit ADC (ADS7823). A comparison between the inference performance of both ADCs is presented at the end of this section.

7.3.1 General Aspects of ADC Conversion

Roughly speaking, an ADC takes the analog signal at its input and produces a value by determining how far the input voltage is between the low and high reference voltages. The more discrete levels an ADC has, the more accurate the digital representation of the analog signal is. An ADC with a larger number of “levels” (i.e., bits) will provide better accuracy and larger resolution than an ADC with fewer bits.

The first ADC employed in the platform is 8-bit ADC (PCF8591). The corresponding circuit schematic is shown in Fig. 7.4. The analog signal is output from the Analog Discovery 2 board using channel 1, connected to the analog input AIN0 of the PCF8591. The SDA and the SCL pins of PCF8591 are connected to the Jetson board’s corresponding pins (I^2C bus 0). By doing so, data can be received in the NVIDIA device from the ADC via I^2C. Note that the grounds of all devices (i.e., Jetson TX2, Analog Discovery 2, and PCF8591) are connected to guarantee the same low reference voltage.

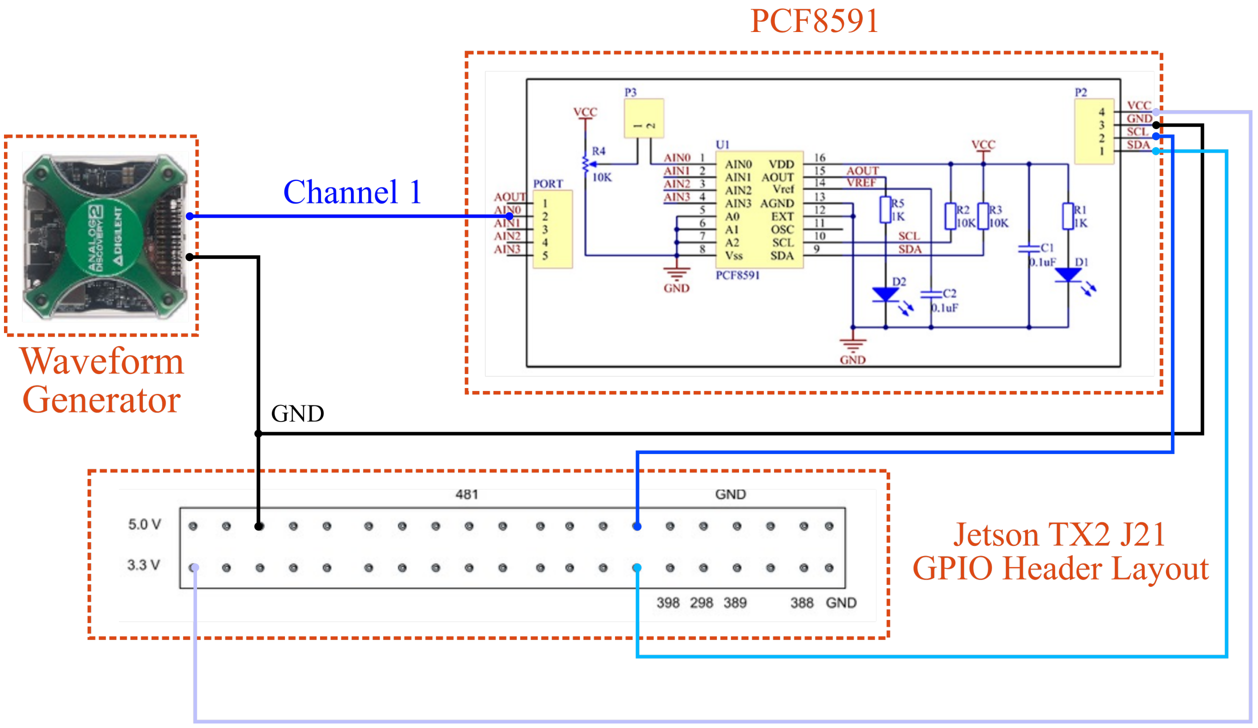


Figure 7.4: Circuit schematic using the PCF8591 ADC.

An example of inference using the PCF8591 is presented in Fig. 7.5. The CNN can detect whether an oscillation is occurring or not with relatively good accuracy[1] (namely, 99.86% in average).

The second ADC variant used in the platform project is a 12-bit ADC: an ADS7823. Logically, the ADS7823 has a better resolution and a higher accuracy than the PCF8591.

[1]It is worth mentioning that physical forced oscillation signals do not have a clear pattern as the sinusoid signal in Fig. 7.5. The accuracy of the CNN downgrades when facing real measurements. However, the method maintains a remarkable inference performance using short data windows and reduced computational time. For further discussion, the reader is referred to [12].

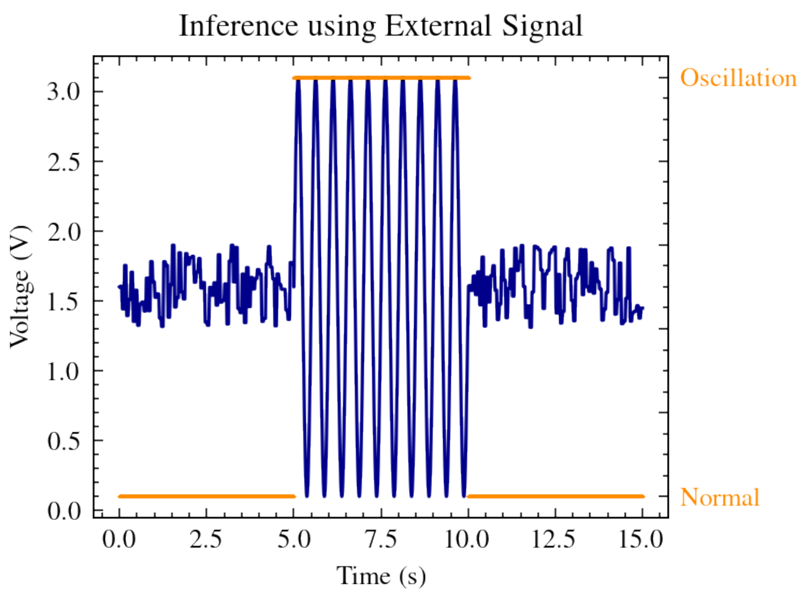


Figure 7.5: Inference result for the implementation of PCF8591 in data acquisition.

The circuit schematic for the ADS7823 is similar to the one in Fig. 7.4 and, therefore, will be omitted. The ADS7823 also uses I^{2}C communication to transmit the signal to the NVIDIA Jetson TX2 board.

7.3.2 A Simple ADC Comparison

We assessed the effect of the ADC resolution by performing 10,000 inferences with the trained CNN model. Results are presented in Table 7.1, where average accuracy is obtained after averaging the accuracy of all inferences over 10,000 experiments. The ADS7823 achieves better performance since the converted signal has a better resolution, and it is easier for the CNN to identify the oscillation patterns and classify the oscillation condition correctly. A more detailed comparison between both ADCs can be seen in Section 7.5.

Table 7.1: Inference accuracy using different ADCs.

ADC	Average Accuracy
ADS7823	0.9986
PCF8591	0.9513

7.4 Experiment Automation

The previous experimental conditions are insufficient to draw significant conclusions about the effect of the ADC selection on inference. By varying other parameters of the test signal, such as frequency and noise level variation, different scenarios can be crafted where we can extract more insight concerning the actual performance differences.

Performing a relevant parameter sweep *manually* is a time-consuming effort. However, one of the advantages of the WaveForms SDK is the possibility of automating several tests using the Python API. This section discusses setting up a simple automation approach by establishing a communications network between the signal generator, the host computer, and the NVIDIA device.

The setup is shown in Fig. 7.6. The communication between the host computer and the Jetson TX2 takes place through TCP/IP socket API. The laptop is configured as the client and the Jetson TX2 as the server. The WaveForms SDK, running on the computer, controls the Analog Discovery 2 board, connected via USB. The output of the signal generator passes through the ADC converter and is read by the TX2 board using I^2C.

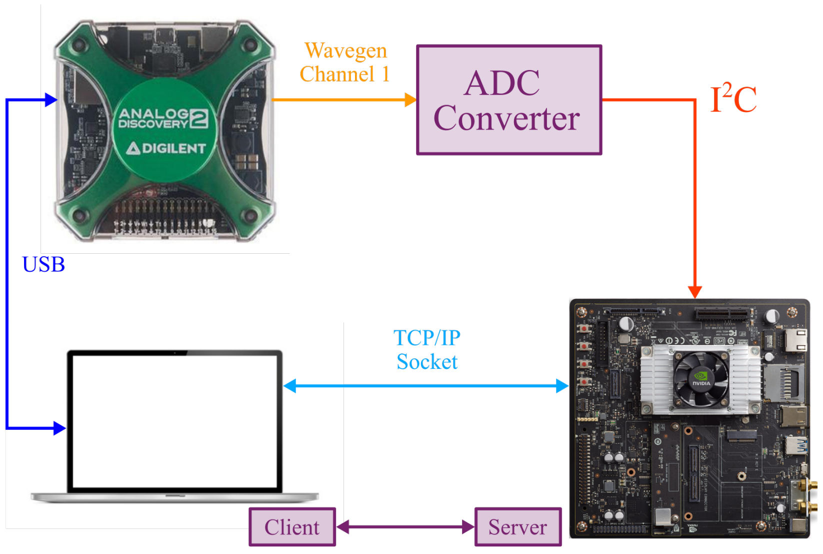


Figure 7.6: Communication map in the proposed experimental setup.

The importance of automation for both testing and communication is illustrated with a simple example. Consider an experiment that consists of a frequency sweep over 100 values. Such variation can be easily created using a loop in the WaveForms SDK. Then, the Analog board outputs the signal to the ADC. However, the host computer must trigger the Jetson

TX2 to start acquiring data. Moreover, the following experiment should begin only once the Jetson TX2 completes inference with the current signal. To achieve this, the laptop and the Jetson are set in a two-way communication link over TCP/IP. The host computer is the client, and the Jetson is the server. The complete workflow is presented in Fig. 7.7.

7.5 Results

A frequency variation experiment is now described after portraying the communication map and the hardware connection. We perform 300 experiments varying the frequency from 1 to 300 Hz, keeping constant the sinusoidal amplitude during the oscillation event. CNN inference is performed on 1000 windows of the produced signal. Half of the windows correspond to normal conditions, and the other half is a sustained oscillation. After the TX2 computes 1000 inferences, a message is sent to the client (host computer) to start the next experiment by varying the signal frequency.

The results of the frequency variation experiment are shown in Fig. 7.8a. The x-axis indicates the frequency value, and the y-axis represents the average accuracy over the 1000 inferences during an experiment. We observe that for low frequencies, the performance of both ADCs is close to each other. Nevertheless, as the frequency increases, the effectiveness of the PCF8591 ADC downgrades rapidly. Despite this, both ADCs are found to be effective in the context of subsynchronous oscillations, where oscillations of concern are below 60 Hz.

Fig. 7.8a is replotted in a logarithmic scale in Fig. 7.8b, which is equivalent to having carried out a logarithmic sweep. We aim to identify the critical frequency in which the performance of the PCF8591 downgrades. We observe that the 8-bit ADC is not effective after $\approx$ 100 Hz. However, even the ADS7823 exhibits a significant reduction in accuracy when the frequency exceeds $\approx$ 200 Hz. Recall that both experiments are using the same CNN model. Therefore, our experiments emphasize the importance of appropriate hardware selection in *every* stage when an ML solution is deployed on a practical application.

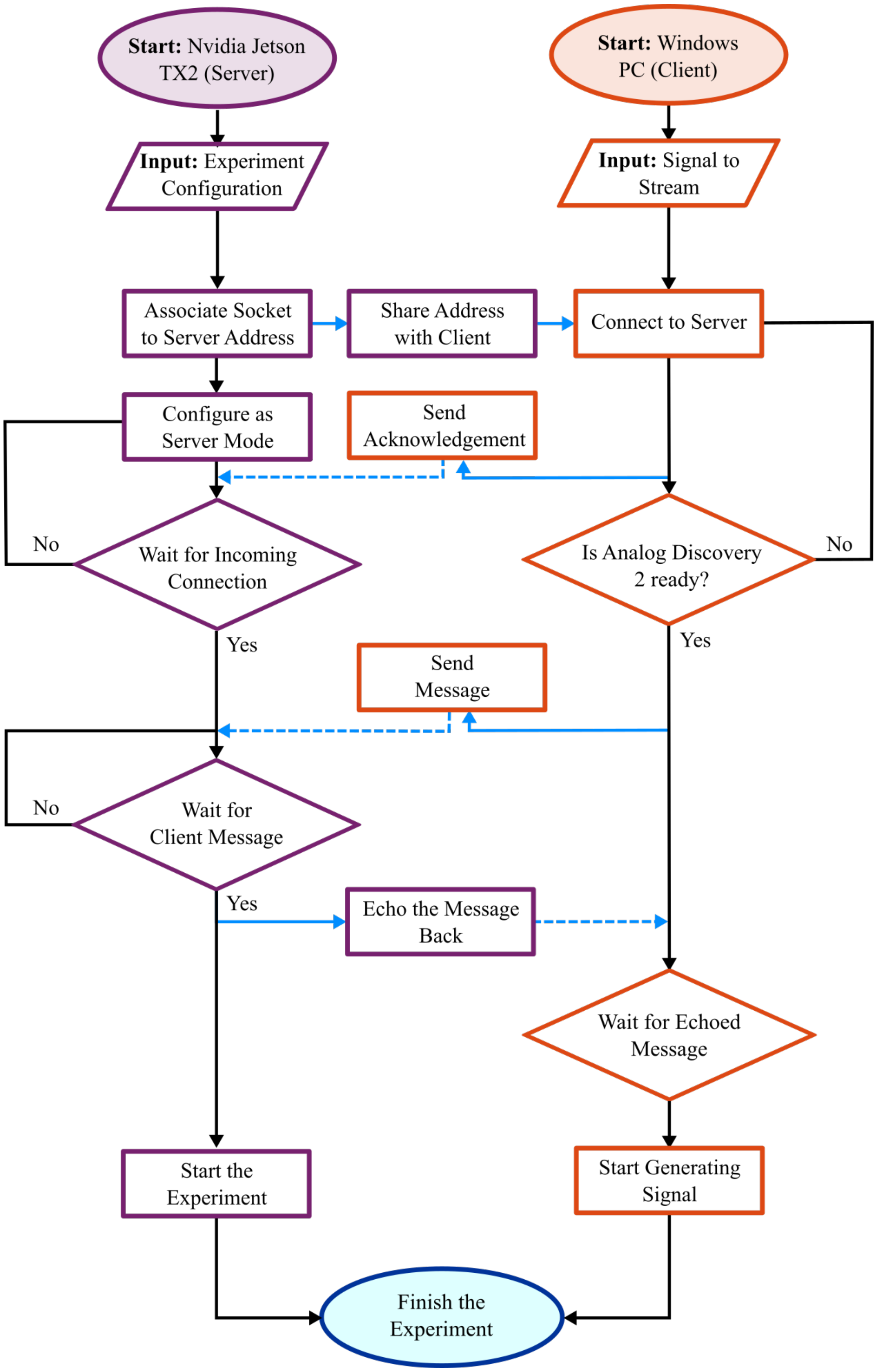
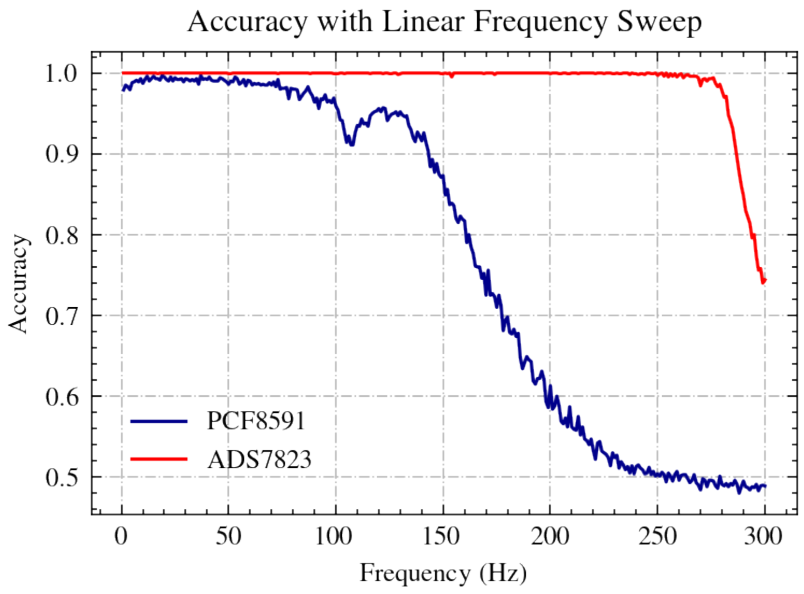
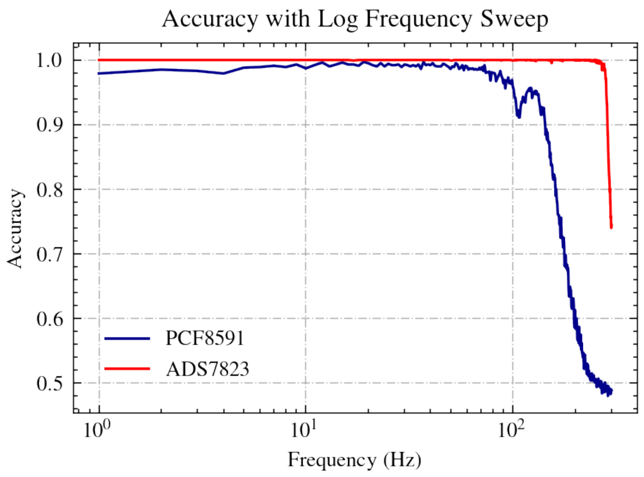


Figure 7.7: Experiment automation workflow. The solid blue lines indicate information sent between server and client, & dashed lines indicate (uncertain) receipt of message.



(a) Linear sweep.



(b) Log sweep.

Figure 7.8: Frequency variation experiment results.

Furthermore, we plot the inference results for all the time windows in a histogram (Fig. 7.9). The x-axis corresponds to the accuracy levels and the y-axis to how many times the CNN achieved the corresponding accuracy. Note that both histograms are skewed towards the 1.0 accuracy, meaning that both ADCs are effective at detecting the oscillation in the particular experiment. However, the PCF8591 bins are distributed horizontally, indicating the accuracy is more sensitive to the resolution of the ADC. In general, we can conclude that the higher the resolution of the ADC, the easier it is for the CNN to detect the oscillation patterns and classify the oscillation condition correctly. The results in Table

7.2 quantify these observations. The reader is referred to [12] for performance analysis of the CNN when subjected to real-world data.

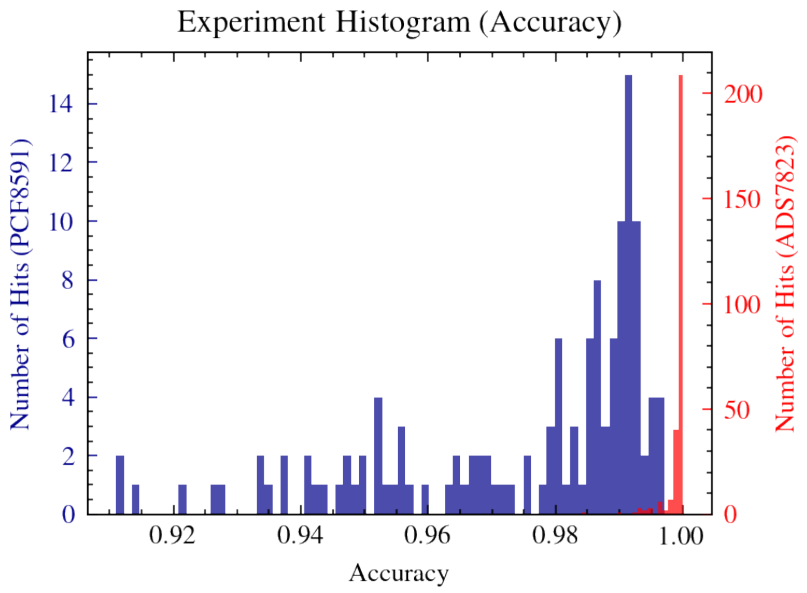


Figure 7.9: Accuracy histogram.

Table 7.2: Statistical results of the frequency variation experiments.

ADC	**Average Accuracy**	
	Mean	**Standard Deviation**
PCF8591	0.97498	0.0213
ADS7823	0.99940	0.00170

7.6 Conclusions

This chapter has presented a low-cost platform for evaluating the real-time performance of CNN models in the detection of sub-synchronous forced oscillation in power grids. Potential applications of such a platform beyond teaching and demonstration are also related to the cheap and fast prototyping of ML-based edge embedded solutions with real-time constraints, such as protective relays.

In our setup, a synthetic signal was generated using a generator (Analog Discovery 2), employed throughout several introductory electronics courses. The signal is converted to a digital representation thanks to an ADC and then transmitted to an NVIDIA Jetson TX2 device. This ML-edge device is capable of executing in real-time a pre-trained CNN model.

Thus, automation on the signal generation and the communication between the Jetson device and the host computer allows simple testing and validation of the accuracy of the proposed solution for oscillation detection.

We also validated the performance impact of ADC selection through simple experiments. Better accuracy is achieved with a high-resolution ADC for the same CNN model. This aspect is not regularly considered in ML training. Still, it is crucial when the CNN is deployed on hardware as part of an IoT solution for monitoring and diagnostics in cyber-physical systems such as the power grid.

CHAPTER 8
MACHINE LEARNING-BASED EDGE APPLICATION FOR DETECTION OF FORCED OSCILLATIONS IN POWER GRIDS

8.1 Introduction

Power system oscillations are generally classified as free and forced. *Free oscillations* appear as the system's response, for instance, to accommodate a change of loads. Free oscillations are *structural* to the system dynamics. In contrast, *forced oscillations* occur when exogenous stimuli with a rich enough spectral component (e.g., cycle-limited control actuation [93], or periodic disturbances [94]) excite the system, thus producing oscillating modes [95]. While different control systems such as the PSS aim at attenuating the effects of free oscillations, forced oscillations could lead to system-wide cascading outages if the excited modes are unstable [96]. The only remedial action is to disconnect the equipment that causes the oscillation or drastically reduce its output power, as in the case of wind farms [83].

Several detection methods have been proposed given the adverse potential of forced oscillations on the grid (e.g., [97]). In [98], a novel multi-delay self-coherence method using data measured by PMUs is designed not only to detect but also to locate the source of a forced oscillation. The reader is referred to [79] for a comprehensive review of such techniques. Meanwhile, [99] provides a suppression control method that can automatically induce a power injection into the power grid to compensate for the impact of the forced oscillation.

Most oscillation detection proposals consist of a chain of signal processing stages such as noise removal and filtering (e.g., [83]). While some of these techniques have proven effective, their computational efficiency is constrained by the complexity of intermediate calculations. Overcoming this time requirement is crucial if such algorithms are deployed as real-time

Portions of this chapter appear in S. A. Dorado-Rojas, S. Xu, L. Vanfretti, M. I. I. Ayachi, and S. Ahmed, "ML-Based Edge Application for Detection of Forced Oscillations in Power Grids," presented at the 2022 IEEE Power & Energy Society General Meeting [12].

detection pipelines. Even more importantly, if they are to be deployed at the edge. Some recent contributions have addressed the online performance of detection methods (e.g., [100], [101]). For instance, the detection algorithm in [85] takes 1.7 s to process and label a forced oscillation. In, [86], the detection time is about 350 ms. From the commercial point of view, [102] presents a set of detection methods including patented solutions now implemented in commercial relays. In summary, it is natural to question if most signal processing-based solutions are also feasible for real-time deployment.

ML algorithms show promising potential as data-driven methods for oscillation detection (see [103], [104]) and efficient real-time deployment while harnessing Graphical Processing Unit (GPU) power. Nowadays, ML models can be easily optimized offline using existing data, which is known as *training.* The trained model can be deployed in IoT devices at the edge to process measurements directly. IoT devices are equipped with a GPU. GPUs allow ML models to be executed in real-time efficiently [105]. Like the NVIDIA Jetson TX2, IoT devices have been proven effective for real-time ML-based solutions. Successful case studies arise from applications such as depth reconstruction from images [106], and face recognition [107] among others. That being said, the purpose of this work is to make the case that ML-based models are feasible solutions for real-time pipelines for forced oscillation detection at the edge, e.g., as part of a new type of protective relay.

Previous works regarding ML for oscillation detection have focused on developing and deploying the algorithm *on a computer*, either for offline or real-time detection. Such "server-centered service" adds the requirement of a communication network for ambient data collection. An example of this is shown in [100] where an oscillation detection method based on an improved XGboost algorithm and random power system measurements is introduced. The trained model is applied to online oscillation detection of a power system, with the algorithm running on a computer. So, if the algorithm were to be deployed for real-time detection, data must be streamed through a communication network. Likewise, in [101], an ML algorithm based on regularized exponential forgetting is proposed. The solution is suitable for non-stationary data analysis. The model is deployed on a conventional computer, so measurements such as currents, voltages, and angle differences must be transmitted through a communication network to apply it to real-time ambient data.

Data transmission from client to server introduces further delays into the overall detection process, and consequently, practically reduces the computational efficiency of any algorithm. Therefore, deploying ML-based oscillation detection algorithms on edge devices becomes more relevant. As an advantage, it bypasses the communication stages directly and can work with measurement data on sites near potential sources (e.g., wind farms [102]). To fill this gap, this chapter introduces a method and presents an ML-based approach for oscillation detection at the edge deployed on an NVIDIA Jetson TX2. The proposed method can detect forced oscillations accurately using real-time measurements directly while efficiently processing the data with the built-in GPU. We must underline that the core of the contribution is proving the potential of edge devices as means for real-time inference.

The chapter is organized as follows: Section 8.2 describes the procedure to train 1D- and 2D-CNNs. In Section 8.3, we present how the trained models are downloaded to an NVIDIA Jetson TX2 for real-time execution using `TensorRT`. The inference results on ambient data from a wind farm streamed in real-time are discussed in Section 8.4. Finally, Section 8.5 concludes the work.

8.2 CNN Model Description and Training

8.2.1 Foundations of CNN

CNNs are models inspired by the human eye's mechanism to extract visual details. CNNs mimic the structure of receptive fields by allocating specialized neurons to detect features in specific parts of an image. So, a CNN swipes an image looking for local similarities and then filters out the high-level or global features [39]. Feature extraction allows CNNs to discriminate raw data into several categories (i.e., similar images have similar characteristics). Therefore, they exhibit outstanding performance on image classification tasks.

We seek to assess the potential of CNNs for forced oscillation detection in two flavors: one-dimensional (1D) and two-dimensional (2D) CNNs. The former architecture uses the measurement data as an input time series. At the same time, the latter sees the ambient data as *images*.

Both CNN models are developed in Python using the TensorFlow framework and the Keras API. As we will see in Section 8.3, TensorFlow offers a direct path to deploy trained models in hardware using the Software Development Kit (SDK) `TensorRT`.

8.2.2 Training and Validation Data Set Construction

To build training and testing data sets, we used measurements from a wind farm in Oklahoma [83]. The information comes from PMU recordings of voltage, current, and

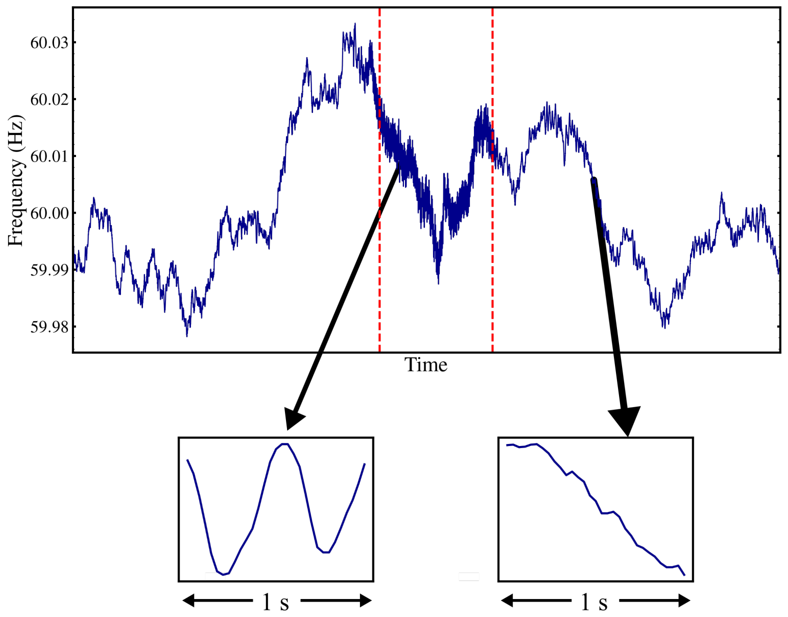


Figure 8.1: Sample of training PMU data (dotted vertical lines indicate the event inset and offset).

frequency during various oscillation episodes, which utility workers classified. In Fig. 8.1, an example event spanning $\approxeq$ 12.5 min = 750 s and features a forced oscillation is shown. We divide the signal into 1 s windows, each window containing exactly 31 samples (there is a one-sample overlap between consecutive windows). By doing so, we generate $\approxeq$ 750 instances from each recording at a particular location, where an instance corresponds to a 1 s window. Examples of training instances can be seen in the 1 s frames in Fig. 8.1.

8.2.3 1D-CNN Model

A 1D-CNN performs a temporal convolution on the input data through several kernels. Besides the dimensionality of the input, the principles behind the operation of a 1D-CNN are the same as conventional 2D-CNNs. This model is included since it is more intuitive to the power engineer because the underlying data set is a one-dimensional time series. Table 8.1 presents a summary of the CNN architecture. All kernels used have a size of 3.

The input is first processed by two convolutional layers (layers 1 and 2), each carrying out a weighted convolution operation with 64 filters. A ReLU function is employed as activation in both layers. After the first two activations, a `maxpooling1D` (layer 3) improves the robustness of the network to noise by effectively decreasing the number of features and selecting the most prominent ones. A `dropout` layer (layer 4) is added right after the pooling

Table 8.1: 1D-CNN model summary.

n_{layer}	**Layer Type**	**Output Shape**	$n_{\text{parameters}}$
1	`conv1d`	`(None, 29, 64)`	256
2	`conv1d`	`(None, 27, 64)`	12352
3	`max_pooling1D`	`(None, 13, 64)`	0
4	`dropout`	`(None, 13, 64)`	0
5	`flatten`	`(None, 832)`	0
6	`dense`	`(None, 100)`	83300
7	`dense`	`(None, 2)`	202
	Trainable Parameters:		96110

layer to prevent further overfitting (i.e., it restricts the network from "memorizing" patterns seen in the training instances). The `flatten` layer (layer 5) converts the multi-dimensional tensor to a one-dimensional vector. This vector is passed to a fully connected layer (`dense`, layer 6) with 100 neurons and a ReLU activation. Finally, another fully connected layer (layer 7) with a softmax activation σ is used to give a probabilistic interpretation to the network output y. Let $\mathbf{z} = [z_1 \quad z_2]^T$ be the input to the last layer, then

$$y = \operatorname{argmax}\left[\sigma\left(\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}\right)\right] = \operatorname{argmax}\begin{bmatrix} \dfrac{e^{z_1}}{e^{z_1}+e^{z_2}} \\ \dfrac{e^{z_2}}{e^{z_1}+e^{z_2}} \end{bmatrix} \tag{8.1}$$

The output encoding is as follows: `1` means an oscillation detected from the input data. At the same time, `0` represents that no oscillation is identified from the passed time window, and thus the power system is safe. The loss function for parameter optimization corresponds to a categorical cross-entropy, as commonly done in classification problems. A visual illustration of the 1D-CNN model is presented in Fig. 8.2. Training and validation results are shown in Fig. 8.3.

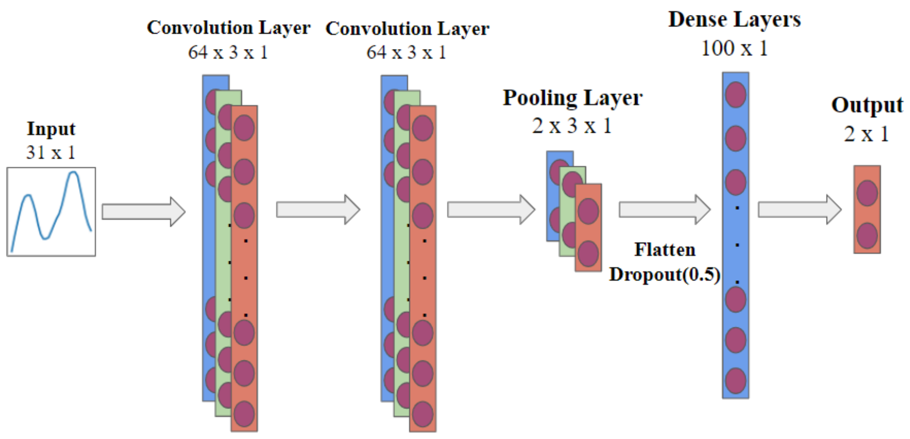


Figure 8.2: Illustration of the 1D-CNN model.

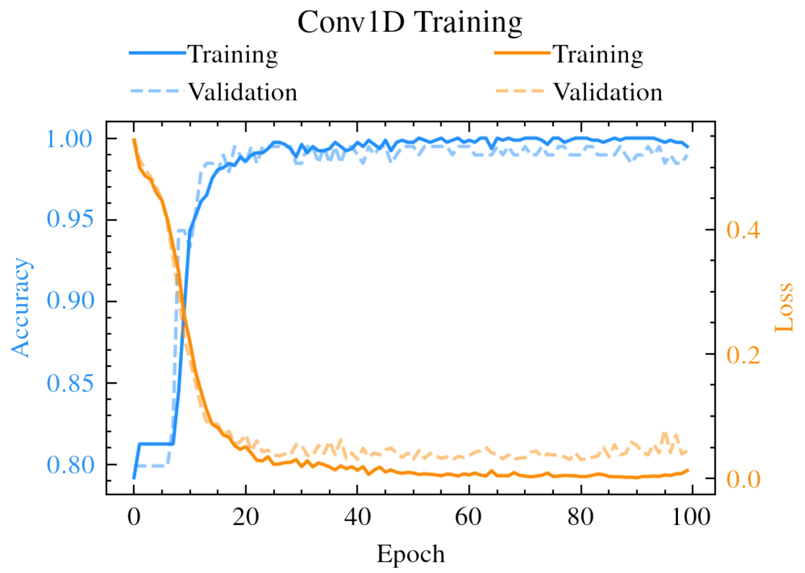


Figure 8.3: Training and validation results for 1D-CNN.

8.2.4 2D-CNN Model

In contrast with the 1D-CNN model, the 2D-CNN model takes images as inputs. The operation, however, is similar to the one of the 1D-CNN network, and thus, a detailed explanation is therefore omitted.

The plots (as RGB images) of the time series data are used as training and validation inputs, such as the one-second windows in Fig. 8.1. The architecture is shown in Fig. 8.4.

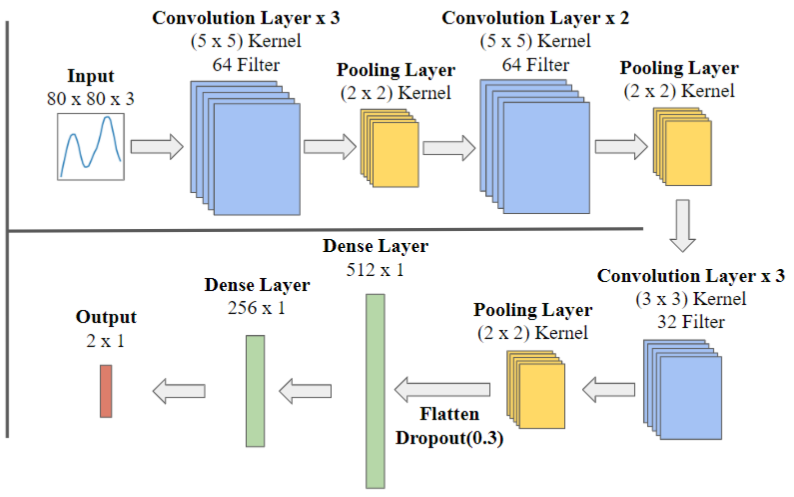


Figure 8.4: Architecture of 2D CNN Model.

By inspection, it is straightforward to notice that the 2D-CNN model is more complex than the 1D-CNN, having a significantly larger number of parameters (cf., $n^{\text{2D-CNN}}_{\text{parameters}} = 2222690$ and $n^{\text{1D-CNN}}_{\text{parameters}} = 96110$) and more layers (cf., $n^{\text{2D-CNN}}_{\text{layers}} = 19$ and $n^{\text{1D-CNN}}_{\text{layers}} = 7$). For this reason, the number of training instances has to be considerably larger to achieve significant performance.

Fortunately, the image-based approach of the 2D-CNN allows performing data augmentation. In this case, random transformations of the images (such as rotations and noising) are carried out to expand the number of training and validation instances, important for this problem due to the low number of training instances and a large number of training parameters. Augmentation is performed automatically by the TensorFlow function `ImageDataGenerator`.

Training and validation results can be seen in Fig. 8.5. Notice that, since the number of parameters is larger, the model needs to be trained for more epochs. For either architecture, given the low number of data instances for this problem, a large number of epochs would be initially thought of as ideal to maximize data utilization. The number of epochs is such that the network finds a “sweet spot” where inference performance is not constrained by overfitting. So, the number of epochs is set to be a maximum of ≈ 300. The resulting model will therefore be used for inference.

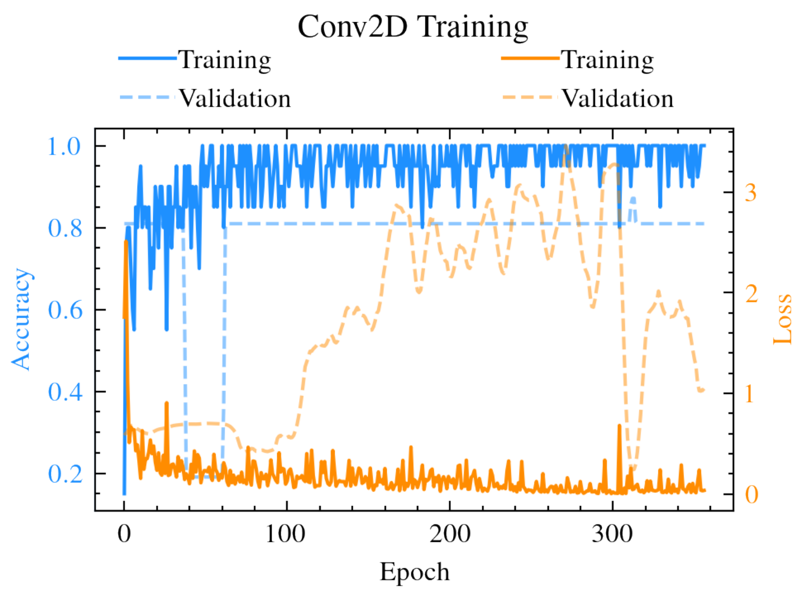


Figure 8.5: Training and validation results for 2D-CNN.

8.3 CNN Models Optimized by TensorRT

Once the CNN models have been trained offline (i.e., on a computer or server), the next step is to prepare them for deployment in a target for real-time inference. The library `TensorRT` is used to convert the model from frameworks such as TensorFlow/Keras and PyTorch to CUDA-compatible code, to be deployed to the target device. The target for our study is the NVIDIA Jetson TX2. For this work, the CNNs were developed in TensorFlow, so the trained models can be exported either as `*.hdf5` files or `*.h5` files. The former format is preferred since fewer intermediate steps are required to import the model within the `TensorRT` library.

The conversion and code optimization process is outlined in Fig. 8.6. As mentioned before, the first step is to save the trained models in a suitable format such as `.hdf5` (for the model weights; so the model would have to be rebuilt and saved) or `.h5` (for the full model). The trained model is then saved to the `TensorRT`-compatible format `.pb`. Once in `TensorRT`, the model could be deployed directly to the target, but an additional optimization process can be carried out to improve real-time performance.

`TensorRT` counts with a set of optimization routines to improve inference performance (i.e., the time it takes to complete inference provided inputs to the model). As part of the actions, while optimizing the code, layers with unused outputs are removed, operations with similar parameters are combined, and subsequent layers are blended into one (for instance,

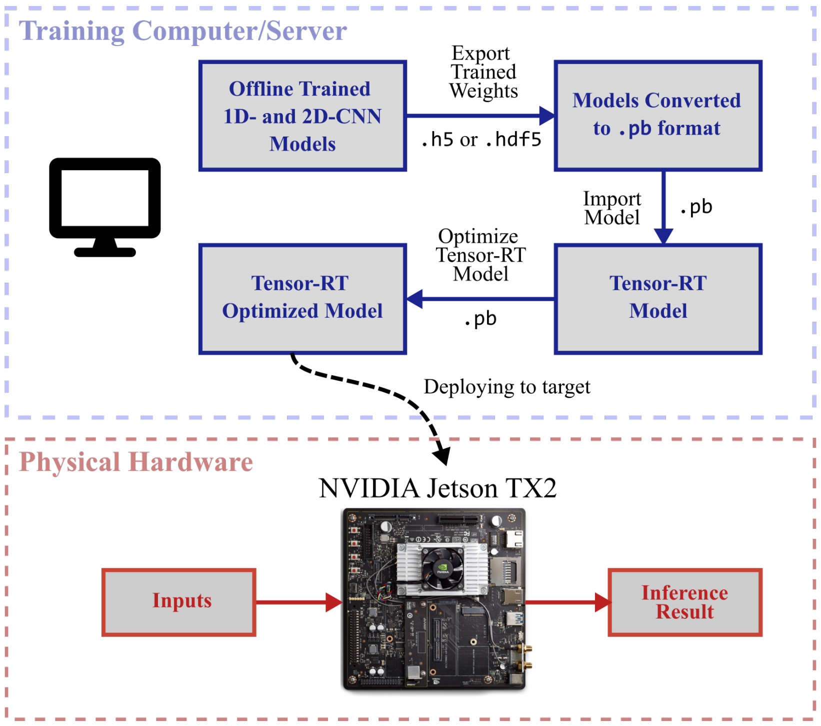


Figure 8.6: Outline of CNN model development for deployment on NVIDIA Jetson TX2 to perform real-time inference.

a convolution operation and an activation function are merged into a single computational layer). Altogether, the optimization routine yields a model with a less computational burden on the target and faster real-time inference.

Table 8.2 shows a comparison of the inference time when the model is deployed on an offline computer and the NVIDIA Jetson device with and without the code optimization routines of `TensorRT`. We observe that, without optimization, the average inference time per sample is within the same order of magnitude for both the Jetson TX2 edge device and the offline computer. However, the edge device infers $\approx$ 10x faster thanks to the optimized code. This result speaks highly of the feasibility of the NVIDIA Jetson TX2 for edge deployment of a real-time oscillation detection tool: the CNNs detect a forced oscillation using 1 s. windows within 10 ms, approximately 3x faster than the algorithm in [86]. Also, note that the inference time for the 2D-CNN is faster than that of the 1D-CNN. This is due to the most extensive use of optimized linear algebra routines in image convolution operations.

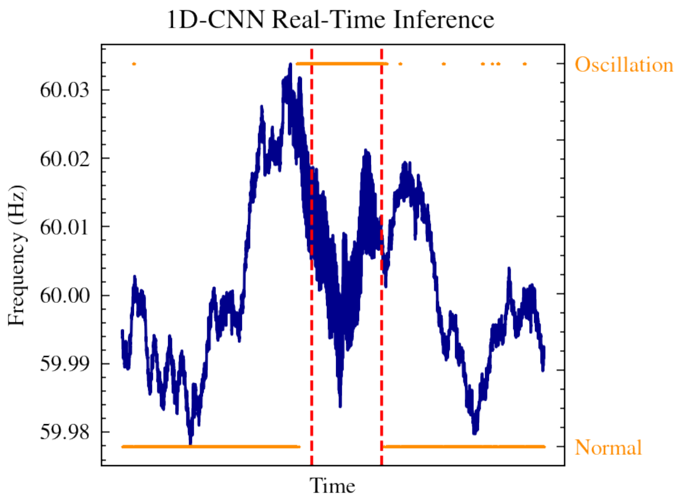
Table 8.2: Average inference time using different hardware devices.

Hardware/Model	1D-CNN	2D-CNN
Windows PC Intel i7-7700HQ 2.80 GHz NVIDIA GeForce GTX 1060 (TensorFlow/Keras Model)	96.931 ms	67.312 ms
NVIDIA Jetson TX2 Non-optimized by TensorRT	74.290 ms	38.379 ms
NVIDIA Jetson TX2 Optimized by TensorRT	**9.787 ms**	**3.410 ms**

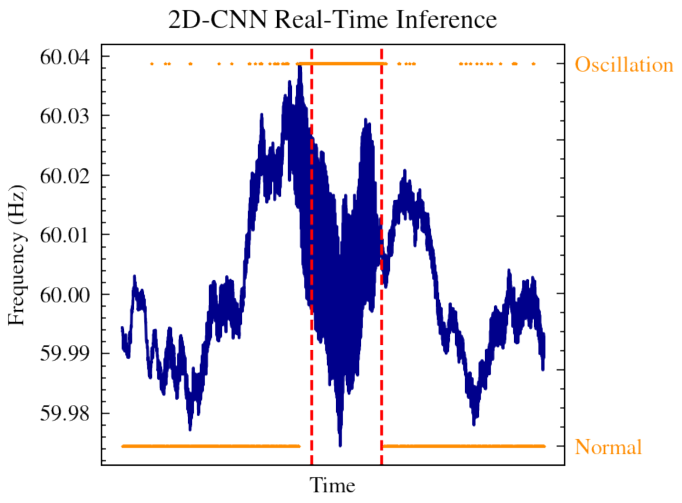
8.4 Detection of Forced Oscillations

The performance of the CNN-based forced oscillation detection method on the NVIDIA Jetson TX2 is evaluated with two different experiments (see Fig. 8.7). On the one hand, ambient data *never seen neither during training nor validation* are used as inputs. On the other hand, synthetic waveforms emulating oscillations created by a signal generator are fed to the NVIDIA Jetson's I2C input ports through a bespoke analog to digital conversion board. Note that the time series require additional preprocessing (for instance, generating the plots from the time series data for the 2D-CNN) in the Jetson's CPU before performing inference in the device's GPU.

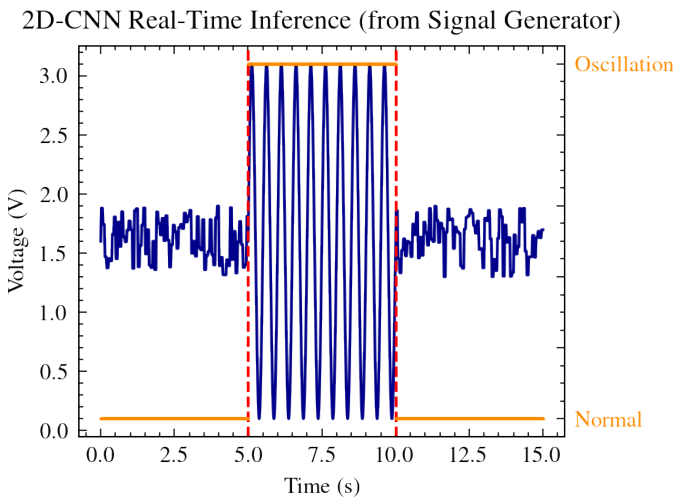
From Figs. 8.7a and 8.7b, we observe that the CNNs succeed at detecting the oscillation and keep providing correct predictions while the event is active. However, accuracy is not 100%, as expected. A simple running window algorithm can be used to discard false positives by keeping track of the CNN recent predictions (e.g., if most of the inferences in the last 2 s are `0`, then the CNN's `1` output should be discarded). Both the 1D- and 2D-CNN detect the oscillations using measurements at different grid locations where they could be deployed.



(a) **1D-CNN output to ambient data.**



(b) **2D-CNN output to ambient data.**



(c) **2D-CNN output to synthetic generator.**

Figure 8.7: Real-time inference results from ambient data and synthetic signal generation.

Fig. 8.7c illustrates how oscillations are detected when an external signal is applied to the Jetson TX2 board using a signal generator. The forced change results from superimposing a sinusoid on a noisy signal. Accuracy improves compared to the ambient data experiment since the signal is not as "challenging" as those from the grid. The result shows that CNNs can learn the patterns of an oscillation using data from one system and then identify such events in another (i.e., *transfer learning* [39]).

8.5 Conclusions

We introduced an ML-based approach for detecting forced oscillations in power grids using an IoT edge device, an NVIDIA Jetson TX2. Two NN models, 1D- and 2D-CNNs, respectively, were trained using the TensorFlow/Keras framework. Then, the trained model code was optimized using the `TensorRT` library for real-time execution at the edge. Optimized code has proven to be faster than offline execution on the hardware. We evaluated the performance of the proposed CNNs on two different experiments: using real-world ambient data and feeding the NN input with oscillation signals created from a signal generator. The pipeline has proven to be a practical and feasible solution for oscillation detection in IoT-based monitoring systems based on the observed results.

CHAPTER 9
ORTHOGONAL LAGUERRE AND LADDER RECURRENT NEURAL NETWORKS

Preliminaries

Discrete-Time Systems

A discrete-time (DT) system is one whose input and output signals are defined at discrete intervals of time. In this context, a DT system constitutes a mapping between sequences represented mathematically by a system of difference equations (DEs).

Let $\mathbf{y} \in \mathbb{R}^{n_y \times 1}$ be the vector of n_y outputs, $\mathbf{u} \in \mathbb{R}^{n_u \times 1}$ the vector of n_u inputs, then a DT system is represented by $\max(m, n)$-th order system of difference equations:

$$\begin{aligned} \mathbf{y}_{k+n} = \bar{\mathbf{h}}\,(&k, \mathbf{y}_{k+n-1}, \mathbf{y}_{k+n-2}, \ldots, \mathbf{y}_k, \\ &\mathbf{u}_{k+m}, \mathbf{u}_{k+m-1}, \mathbf{u}_{k+m-2}, \ldots, \mathbf{u}_k)\,. \end{aligned} \tag{9.1}$$

It is possible to rewrite the system in (9.1) as a set of DEs, which is known as a *state-space* representation. This is founded on the concept of state. The state of a system at time $k = k_0$ is "the information that, together with the input $\mathbf{u}$ for $k \geq k_0$ determines uniquely the output $\mathbf{y}$ for $k \geq k_0$" [108]. A state-space representation for a DT system is given by

$$\begin{aligned} \mathbf{x}_{k+1} &= \bar{\mathbf{f}}\,(k, \mathbf{x}_k, \mathbf{u}_k) \qquad \text{(State Equation)} \\ \mathbf{y}_k &= \bar{\mathbf{g}}\,(k, \mathbf{x}_k, \mathbf{u}_k) \quad \text{(Output Equation)}. \end{aligned} \tag{9.2}$$

The function $\bar{\mathbf{f}}$ represents an updating policy since it moves the states one step forward in time. $\bar{\mathbf{g}}$ is a static nonlinear operator. The state vector of a system gives insight into the internal operation of the system. Therefore, state-space descriptions are called *grey-box* models.

Portions of this chapter appear in S. A. Dorado-Rojas, B. Vinzamuri, and L. Vanfretti, "Orthogonal Laguerre Recurrent Neural Networks," presented at the 34th Conference on Neural Information Processing Systems (NeurIPS) [13].

In particular, the case where the forward mapping is linear can be represented by linear transformations. Furthermore, if time does not appear explicitly in the equations, we have a discrete-time linear time invariant (DTLTI) system

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \qquad &\text{(State Equation)} \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k \qquad &\text{(Output Equation).} \end{aligned} \tag{9.3}$$

Note that the output is not directly connected to the input. This is a reasonable assumption for most practical DT systems.

9.1 Introduction

RNNs are special structures employed to deal with structured data such as sequences and time series. RNNs have been successful in tasks such as natural language processing, forecasting, and speech-to-text recognition. By definition, the output of an RNN at time k ($\mathbf{y}_k$) depends not only on the current input $\mathbf{u}_k$ but also on past values of the output $\mathbf{y}_{k-1}$. The current output is generally computed as

$$\begin{aligned} \mathbf{h}_k &= \boldsymbol{\sigma}\left(\mathbf{u}_k, \mathbf{y}_k, \mathbf{h}_{k-1}\right) \\ \mathbf{y}_k &= \boldsymbol{\gamma}\left(\mathbf{h}_k\right) \end{aligned} \tag{9.4}$$

where $\boldsymbol{\sigma}$ and $\boldsymbol{\gamma}$ are activation functions. For a vanilla RNN, the hidden state is computed as $\mathbf{h}_k = \boldsymbol{\sigma}\left(\mathbf{W}_u\mathbf{u}_k + \mathbf{W}_y\mathbf{y}_{k-1}\right)$ with output $\mathbf{y}_k = \mathbf{h}_k$. This structure has been shown to have training difficulties such as unstable gradients and long-term memory vanishing [39].

Long Short-Term Memory (LSTM) cells and Gated Recurrent Unitss (GRUs) have been developed to overcome the main drawbacks of vanilla RNNs. However, they have been shown to suffer long-term memory issues when faced with sequences with more than 1000 samples [109], [110]. For this reason, several novel architectures have been recently formulated as an attempt to address the caveats of LSTMs and GRUs. Most of these proposals have been constructed following either the LSTM approach (i.e., by decoupling the state in two vectors) or the Legendre Memory Unit (LMU) structure (i.e., by performing gating operations to preserve only one hidden state).

The introduction of LMUs represents a deep connection between RNNs and dynamical systems from the *design* point-of-view, despite some previous efforts to *analyze* the structure from the dynamical systems perspective [111], [112]. The memory update policy of LMUs is the state equation of a DTLTI system described by

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}u_k \tag{9.5}$$

where $\mathbf{x} \in \mathbb{R}^{n\times 1}$ is the memory (state of the cell), u_k is the input to the memory gate, and the pair $(\mathbf{A}, \mathbf{B})$ corresponds to a discretized state-space representation of shifted Legendre polynomials, portrayed by n coupled ordinary differential equations (ODEs). The orthogonal time-domain projection inside LMUs improves memory capabilities, reduces the number of trainable parameters, and makes the network capable of achieving state-of-the-art performance on challenging benchmark tasks for sequential data classification such as psMNIST [113].

Nevertheless, LMUs *only* make use of the state equation. The output equation (i.e., $\mathbf{y}_k = \mathbf{C}\mathbf{x}_k$) does not appear within their structure. The use of this equation inside a RNN becomes even more relevant if the $(\mathbf{A}, \mathbf{B})$ pair represents an orthogonal family of functions, like shifted Legendre polynomials.

With an embedded full state-space representation based on orthogonal functions, a RNN would be able to learn *any* stable DTLTI dynamics [114]. In other words, the use of the output equation would enable the RNN to steer the behavior of some internal signal towards *any* energy-limited signal, encoded inside the DTLTI system represented by the state equation and the output equation. Thus, the generalization capabilities of the network would be increased by including an orthonormal basis in the state equation, and the explicit use of the output equation.

9.1.1 Problem Specification

The most general framework to model physical dynamical systems is that of hybrid systems. A *hybrid system* is one that shows both continuous-time (CT) behavior, described by a system of ODEs, and DT behavior, modeled by a system of DEs. In spite of proposals to understand RNNs from a hybrid [115] or a CT system [116] viewpoint, a DT system framework is still a more familiar fashion to treat RNNs as dynamical systems.

The representation of DT dynamical systems has a close connection to RNNs. According to (9.4), the output of an RNN depends not only on the current input but also on past values of the output and the hidden state. A direct comparison between (9.4) and (9.2), with the state-equation delayed by one sample, shows the hidden state layer can be thought of as the state equation of a nonlinear dynamical system:

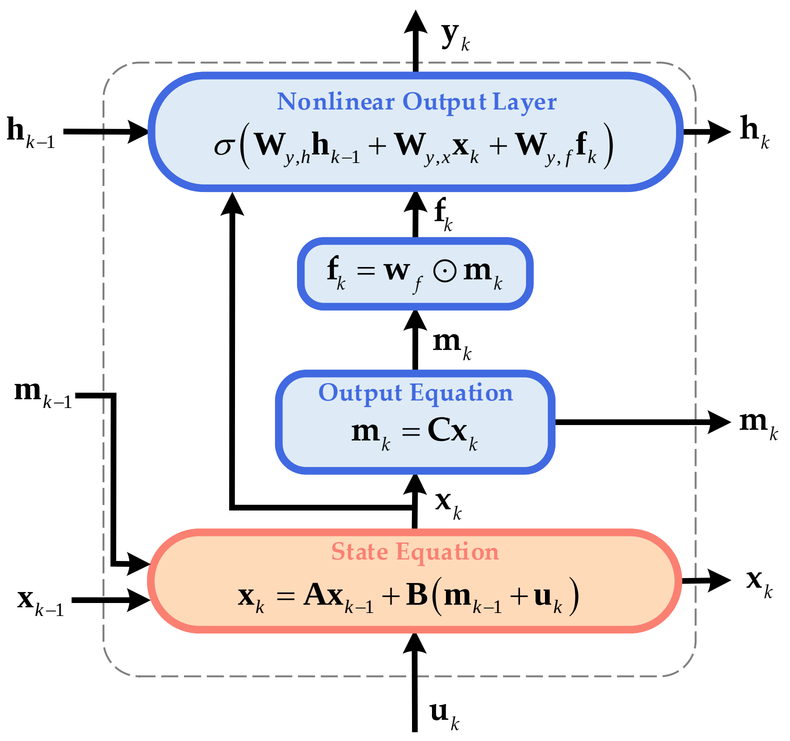
$$\begin{aligned} \mathbf{x}_k &= \bar{\mathbf{f}}\left(k, \mathbf{x}_{k-1}, \mathbf{u}_{k-1}\right) & \longleftrightarrow \quad \mathbf{h}_k &= \boldsymbol{\sigma}\left(\mathbf{u}_k, \mathbf{y}_k, \mathbf{h}_{k-1}\right) \\ \mathbf{y}_k &= \bar{\mathbf{g}}\left(k, \mathbf{x}_k, \mathbf{u}_k\right) & \longleftrightarrow \quad \mathbf{y}_k &= \boldsymbol{\gamma}\left(\mathbf{h}_k\right). \end{aligned} \tag{9.6}$$

This work uses this connection as a starting point to design a memory updating policy for a RNN architecture inspired by the state-space representation of a DTLTI system (Eq. (9.3)). The linear time invariant (LTI) dynamics are encoded using Laguerre polynomials, an orthonormal basis for stable LTI systems. By setting up such a basis in the state equation, the RNN can steer the internal behavior of the embedded DT system to any stable LTI dynamics by learning the corresponding coordinate vector. In particular, for the Ladder network the input-output behavior is selected to match discrete-time delays which help improve the memory capabilities of the RNN constraining its operation by a dynamical system. In other words, Laguerre polynomials are used because of their orthonormal constructive properties rather than by its geometrical implementations.

The contributions of this chapter are the following:

- we propose a novel RNN architecture inspired by DTLTI dynamical systems, called an orthogonal Laguerre RNN.
- we introduce two variants based on the proposed network architecture: a general orthogonal Laguerre network, and a Ladder network which takes advantage of a special case of Laguerre functions that correspond to delay networks;
- we validate the performance of the proposed architecture and its two variants with a system identification and a response prediction benchmark against state-of-the-art RNN architectures.

The architecture (Fig. 9.1) consists of three main pipelines: a state flow $\mathbf{x}$, a memory flow $\mathbf{m}$, and a hidden output flow $\mathbf{h}$. Trainable weights are $\mathbf{w}_f$, $\mathbf{W}_{y,h}$, $\mathbf{W}_{y,x}$ and $\mathbf{W}_{y,f}$. $\mathbf{C}$ might be trainable or not depending on the architecture: for Laguerre, it is trainable whereas for Ladder it is fixed.



$\mathbf{y}_k$
Nonlinear Output Layer
$\sigma\left(\mathbf{W}_{y,h}\mathbf{h}_{k-1} + \mathbf{W}_{y,x}\mathbf{x}_k + \mathbf{W}_{y,f}\mathbf{f}_k\right)$
$\mathbf{h}_{k-1}$
$\mathbf{h}_k$
$\mathbf{f}_k$
$\mathbf{f}_k = \mathbf{w}_f \odot \mathbf{m}_k$
$\mathbf{m}_k$
Output Equation
$\mathbf{m}_k = \mathbf{C}\mathbf{x}_k$
$\mathbf{m}_{k-1}$
$\mathbf{m}_k$
$\mathbf{x}_k$
State Equation
$\mathbf{x}_k = \mathbf{A}\mathbf{x}_{k-1} + \mathbf{B}\left(\mathbf{m}_{k-1} + \mathbf{u}_k\right)$
$\mathbf{x}_{k-1}$
$\mathbf{x}_k$
$\mathbf{u}_k$

Figure 9.1: Orthogonal Laguerre RNN architecture.

This model has an embedded state-space representation of LTI dynamics based on Laguerre functions. The state equation dictates the policy update of the state $\mathbf{x}$. The memory $\mathbf{m}$ is computed as a linear transformation of the state $\mathbf{x}$ through the output equation. The output of the cell is computed by the projection of the previous output, actual input, and updated memory through a nonlinear static layer, in a similar fashion to Non-saturating Recurrent Units (NRUs) [117].

The Laguerre network uses a fixed Laguerre representation $(\mathbf{A}, \mathbf{B})$ to perform an orthogonal state update. The output equation is used so that the network can determine what dynamics are most convenient to compute the memory $\mathbf{m}$ by learning the coordinate matrix $\mathbf{C}$. On the other hand, the Ladder network a specific case of a Laguerre network. It is founded on the relationship between discrete-time Laguerre networks and Kronecker-delta impulses. In this case, the state-space representation is fixed (i.e., the coordinate matrix is not learned) so that the discrete-time behavior inside the RNN structure is imposed by design. By doing so, the RNN becomes able to retain memory information for arbitrary periods of time thanks to the implicit internal delay dynamics.

This chapter is structured as follows: Section 9.2 resumes the most recent contributions in terms of RNN architectures regarding orthogonality and dynamical systems-inspired design. In Section 9.3, the theory underlying the architecture is presented and the design is explained. Experiments are discussed in Section 9.4, Finally, the work is concluded in Section 9.5 with a discussion of the experimental results.

9.2 Related Work

Gated Orthogonal Recurrent Units (GORUs) can be considered as an example of a GRU-inspired solution to the major RNN drawbacks. GORUs are based on the projection of the previous hidden state through an orthogonal matrix together with a modified non-saturated activation to compute the unit output [118]. The orthogonality approach was first introduced in Unitary Evolution Neural Networks, where the hidden output weight matrix is constrained to be a unitary matrix with normal eigenvalues (i.e., magnitude one) to tackle the vanishing gradient problem directly [109]. Another proposal is the scaled Cayley orthogonal recurrent neural network (scoRNN) [119] which reduces the number of trainable parameters by a skew-symmetric matrix parametrization of the weight matrices and avoids complex values during training. Moreover, Light-GRUs (Li-GRUs) are founded on the removal of the reset gate of the standard LMU architecture with a modified ReLU activation [120].

Another GRU-inspired architecture that incorporates some concepts of dynamical systems is the bistable recurrent cell family which has been introduced in two variants: Bistable Recurrent Cell (BRC) and Neuromodulated Bistable Recurrent Cell (nBRC) [121]. In this case, the hidden state is updated according to a nonlinearity that forces two stable states on the underlying dynamical system. For BRCs, the hidden state is updated using local information only (i.e., of each cell), whereas, for nBRCs, the state is updated using the information of all cells in a layer.

In contrast with approaches with a single hidden state, JANET is derived from the LSTM structure to result in an architecture that enhances the role of the forget gate by removing the input and output gates [122]. Likewise, NRUs decompose the state into a memory and a cell state. The memory is updated through vector operations on the previous cell by writing and erasing content in specific directions [117]. The output is computed using a nonlinear layer using the previous output, the updated memory, and the cell input. This output layer configuration is also seen in LMUs. For LMUs, in [113], a memory update is proposed using an orthogonal decomposition using shifted Legendre polynomials, an $\mathcal{L}_2$

orthogonal basis.

Unlike LMUs, our Laguerre and Ladder architectures use a complete state-space representation to embed LTI dynamics inside an RNN. This guarantees that any stable LTI system can be represented by the appropriate coordinate vector inside the network. Moreover, shifted Legendre polynomials, the orthonormal basis used in LMUs, is obtained from Legendre polynomials after time scaling and time shifting operations. The implementation of LMUs has shown that the time scaling must be set during network construction depending on the sequence to be processed by the RNN. Laguerre functions are orthonormal in $[0, \infty)$. So, no further scaling or shift must be done to implement them when constructing the RNN.

We can see that the Laguerre functions are selected because of their capability to span a vector space (i.e., the space of stable LTI systems). This contrasts with the motivation of GORUs where orthonormal matrices were used to preserve information under a linear transformation between spaces. Hence, orthonormality is used in the algebraic rather than in the geometrical sense.

9.3 Ladder and Laguerre Orthogonal Architecture

This section discusses the theory behind Laguerre functions from the dynamical systems viewpoint before detailing the proposed RNN architecture and its two variants. Special emphasis is made in the state-space representations of Laguerre polynomials in CT and DT. It is shown that the DT transfer function representation of the Laguerre networks leads to a special relationship between Laguerre functions and shifted unit impulses. Due to this, it is possible to embed delay dynamics inside an RNN. This latter relationship is the basis for the formulation of the Ladder architecture.

9.3.1 Laguerre Functions

Laguerre functions are a family of Eigen functions arising from the Sturm-Liouville problem characterized by their orthonormality. The set of Laguerre functions $\ell^{(i)}(t)$ $(i = 1, 2, 3, \dots)$ is defined as

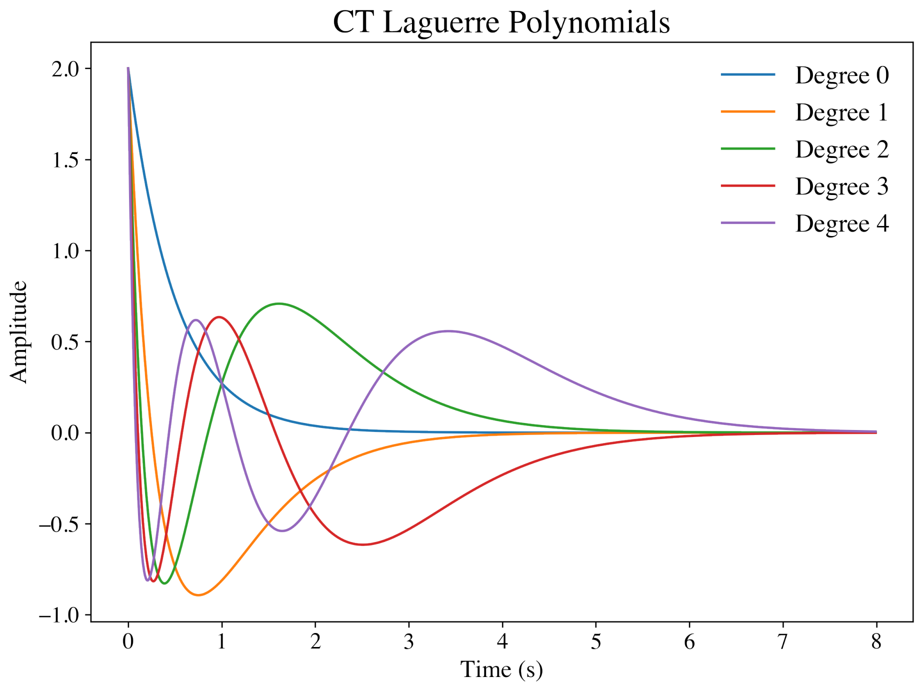


Figure 9.2: CT Laguerre polynomials.

$$\begin{aligned}\ell_1(t) &= \left(\sqrt{2p}\right) e^{-pt} \\ \ell_2(t) &= \left(\sqrt{2p}\right)(1-2pt)\, e^{-pt} \\ &\vdots \\ \ell_i(t) &= \left(\sqrt{2p}\right) \frac{e^{pt}}{(i-1)!} \frac{d^{i-1}}{dt^{i-1}} \left[t^{i-1} e^{-2pt}\right] \\ &\vdots\end{aligned} \tag{9.7}$$

where p is a parameter called time scaling factor.

Laguerre functions are a complete set over $[0, \infty)$. Thus, they can be used to reconstruct any $\mathcal{L}_2$ function $f(t)$ by means of a Generalized Fourier Series expansion

$$f(t) = \sum_{i=0}^{\infty} c_i \ell^{(i)}(t) \tag{9.8}$$

with c_i being the ith coefficient of f on the ith element of the Laguerre basis. Moreover, under mild assumptions on the function $f \in \mathcal{L}_2$, a Generalized Fourier Series expansion with N Laguerre functions can be used to define an arbitrarily close approximation to the function f

$$\int_0^\infty \left(f(t) - \sum_{i=1}^{N} c_i \ell^{(i)}(t) \right) dt < \varepsilon \tag{9.9}$$

for any $\varepsilon > 0$ [123]. This characteristic means that it is possible to represent dynamics whose characteristics are conveniently captured by an absolutely integrable signal using a finite number of Laguerre functions [124]. Systems completely characterized by an absolute integrable signal correspond to stable LTI systems. Then, it is possible to represent LTI dynamics by Laguerre polynomials.

Laguerre polynomials have a convenient mathematical representation in state-space [123]. Let $\mathbf{l}$ be the vector containing the first N Laguerre polynomials as functions of time $\mathbf{l} = \begin{bmatrix} \ell_1(t) & \ell_2(t) & \dots & \ell_N(t) \end{bmatrix}^T$.. In particular, we are interested in the values at $t = 0$, that is, $\mathbf{l}(0) = \begin{bmatrix} \ell_1(0) & \ell_2(0) & \dots & \ell_N(0) \end{bmatrix}^T$. Then, the Laguerre polynomials can be generated by

$$\mathbf{l} = e^{\mathbf{A}_\ell t}\mathbf{l}(0) \tag{9.10}$$

where the matrix $\mathbf{A}_\ell$, referred to as Laguerre matrix, is parametrized in terms of the scaling factor p as follows:

$$\mathbf{A}_\ell = \begin{bmatrix} -p & 0 & \dots & 0 \\ -2p & -p & \dots & 0 \\ \vdots & \dots & \ddots & \vdots \\ -2p & -2p & \dots & -p \end{bmatrix}. \tag{9.11}$$

The Laguerre matrix is lower triangular. Moreover, it is parametrized in terms of p only. This mathematical convenience supports its application in areas such as Model Predictive Control [123], [125].

The vector of initial conditions together with the Laguerre matrix can be used to constitute a state-space representation for any stable CTLTI system [123]. For simplicity, consider the single-input single-output case

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}_\ell \mathbf{x} + \mathbf{B}_\ell u \\ y &= \mathbf{C}_\ell \mathbf{x} \end{aligned} \tag{9.12}$$

with $\mathbf{A}_\ell \in \mathbb{R}^{n \times n}$, $\mathbf{B}_\ell = \mathbf{l}(0)$, and $\mathbf{C}_\ell = [c_1 \;\; c_2 \;\; \ldots c_n]$. $\mathbf{c}$ is a coefficient matrix that depends on the system being considered. This equation can be extended to multiple inputs by concatenating the initial condition vector along the columns, and computing the coefficient vector for each of the output channels. Either way, the matrices $\mathbf{A}_\ell$ and $\mathbf{B}_\ell$ are fixed for all stable CTLTI systems. The coefficient matrix, which changes from system to system, represents the coefficients of specific dynamics on the Laguerre basis.

9.3.2 Discrete Laguerre Functions and Unit Delays

A DT Laguerre network $\ell^{(i)}(z)$ is generated from the discretization of a CT Laguerre network, which in turn can be defined as the Laplace transform of the corresponding Laguerre polynomial $\ell^{(i)}(s) = \mathcal{L}\left\{\ell^{(i)}(t)\right\}$.

Let $\ell_k^{(i)} = \mathcal{Z}^{-1}\left\{\ell^{(i)}(z)\right\}$ be the inverse z-transform of $\ell^{(i)}(z)$ (i.e., the time-domain representation of the ith Laguerre network). DT Laguerre sequences are orthonormal over $k \in [0, \infty)$, so they can be employed as a basis to reconstruct any absolutely summable sequence (i.e., any stable linear time-invariant system or finite-energy deterministic signal).

The transfer functions of the DT Laguerre networks are given by

$$\begin{aligned}
\ell^{(1)}(z) &= \frac{\sqrt{1-a^2}}{1-az^{-1}} \\
\ell^{(2)}(z) &= \frac{\sqrt{1-a^2}}{1-az^{-1}}\left(\frac{z^{-1}-a}{1-az^{-1}}\right) \\
&\;\vdots \\
\ell^{(N)}(z) &= \frac{\sqrt{1-a^2}}{1-az^{-1}}\left(\frac{z^{-1}-a}{1-az^{-1}}\right)^{N-1} \\
&\;\vdots
\end{aligned} \tag{9.13}$$

where a is a parameter known as *scaling factor* that represents the location of the Laguerre network pole [123]. Consequently, stability and non-alternating behavior requires $0 \leq a < 1$.

While RNN design from DT Laguerre networks would need special tuning to avoid numerical implementation issues, they have a particular representation which is of fundamental importance for the formulation of the Ladder network in Section 9.3.3. The Ladder network is founded upon a special case of Laguerre functions: discrete-time delays.

The dynamic behavior of delays has received particular attention by the deep learning community given their inherent long-term memory characteristics [126]. However, a practical implementation of a CT delay requires a rational function approximation, usually through Padé approximants as done in [126]. In contrast, DT delays have a simple transfer function representation and a finite-dimensional state-space representation which makes them suitable for an RNN implementation.

To illustrate the connection between Laguerre functions and delay systems, consider the Nth Laguerre network given by

$$\ell^{(N)}(z) = \frac{\sqrt{1-a^2}}{1-az^{-1}}\left(\frac{z^{-1}-a}{1-az^{-1}}\right)^{N-1}.$$

Let $a = 0$. The corresponding Laguerre Nth-order network transfer function is written as $\bar{\ell}^{(N)}(z)$. It can be easily shown that $\bar{\ell}^{(N)}(z)$ has the form

$$\bar{\ell}^{(N)}(z) = \left(z^{-1}\right)^{N-1} = \frac{1}{z^{N-1}} \tag{9.14}$$

which corresponds to the transfer function of a delay of $N-1$ samples. This transfer function is rational and strictly proper. As a consequence, it has a finite-dimensional state-space realization which contrasts with the infinite dimensionality of its CT counterpart. This makes a DT delay suitable for practical implementation.

The inverse z-transform of $\bar{\ell}^{(N)}(z)$ corresponds to the shifted Kronecker-delta impulse

$$\bar{\ell}_k^{(N)} = \delta\left[k-(N-1)\right] = \mathcal{Z}^{-1}\left\{\bar{\ell}^{(N)}(z)\right\}. \tag{9.15}$$

Thanks to this simple equivalence, it is evident that Kronecker-delta impulses can be represented using Laguerre sequences. This is important since shifted Kronecker-delta impulses are the time-domain representation of ideal DT delays. Thus, the use of Laguerre networks inside an RNN can lead to the inclusion of delay dynamics inside an RNN.

The Ladder network is founded on the Laguerre representation of a multiple-input multiple-output system which delays each channel of an input signal by an arbitrary number of samples at each time. For the sake of simplicity, consider a discrete-time system having n_u inputs and $n_y = n_u$ outputs. The transfer function of this squared delay system with $n_u = n_y$, $\mathbf{G}(z) \in \mathbb{C}^{n_u \times n_u}$, is given by

$$\mathbf{G}(z) = \begin{bmatrix} \frac{1}{z^{n_1}} & 0 & \dots & 0 \\ 0 & \frac{1}{z^{n_2}} & \dots & 0 \\ \vdots & \dots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{z^{n_u}} \end{bmatrix} \tag{9.16}$$

where $n_1, n_2, \dots, n_u \in \mathbb{N}$ are the delay samples of each channel. A particular case of this squared delay system is one in which the delays increase unitwise per channel:

$$\mathbf{G}(z) = \begin{bmatrix} \frac{1}{z} & 0 & \dots & 0 \\ 0 & \frac{1}{z^2} & \dots & 0 \\ \vdots & \dots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{z^{n_u}} \end{bmatrix}. \tag{9.17}$$

9.3.3 Architecture Design

The proposed orthogonal architecture is detailed in Fig. 9.3. The blocks with trainable weights are colored in blue (output equation, memory filtering and nonlinear output layer). Note that the memory is constrained to have the same dimensions as the input so that it can be used as means to store input information in the output of the dynamical system.

The design is interpreted as follows: consider an input with n_u features ($\mathbf{u}_k \in \mathbb{R}^{n_u \times 1}$). The state the output equation constitute a DTLTI system that processes the past memory $\mathbf{m}_{k-1}$, state $\mathbf{x}_{k-1}$, and the current input $\mathbf{u}_k$ to update the state $\mathbf{x}_k$ by

$$\mathbf{x}_k = \mathbf{A}\mathbf{x}_{k-1} + \mathbf{B}(\mathbf{u}_k + \mathbf{m}_{k-1}) \tag{9.18}$$

where $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times n_u}$ are discrete Laguerre matrices obtained after discretization of the continuous Laguerre matrices in (9.12). n is the order of the network that corresponds to the number of Laguerre polynomials used to encode the DTLTI dynamics into the RNN.

Once the state is updated, it is fed into the output equation to compute the memory update $\mathbf{m}_k$ through the learnable coordinate matrix $\mathbf{C}$. The memory is filtered elementwise by the local weight vector $\mathbf{w}_f$ to produce $\mathbf{f}_k$ which is fed into the nonlinear output layer.

The nonlinear output layer has been adapted from the one proposed by [117]. It produces n_y hidden outputs through $\mathbf{h}_k = \mathbf{y}_k \in \mathbb{R}^{n_y \times 1}$. The logits are computed by the linear combination of the previous hidden output $\mathbf{h}_{k-1}$, the updated state $\mathbf{x}_k$ and memory

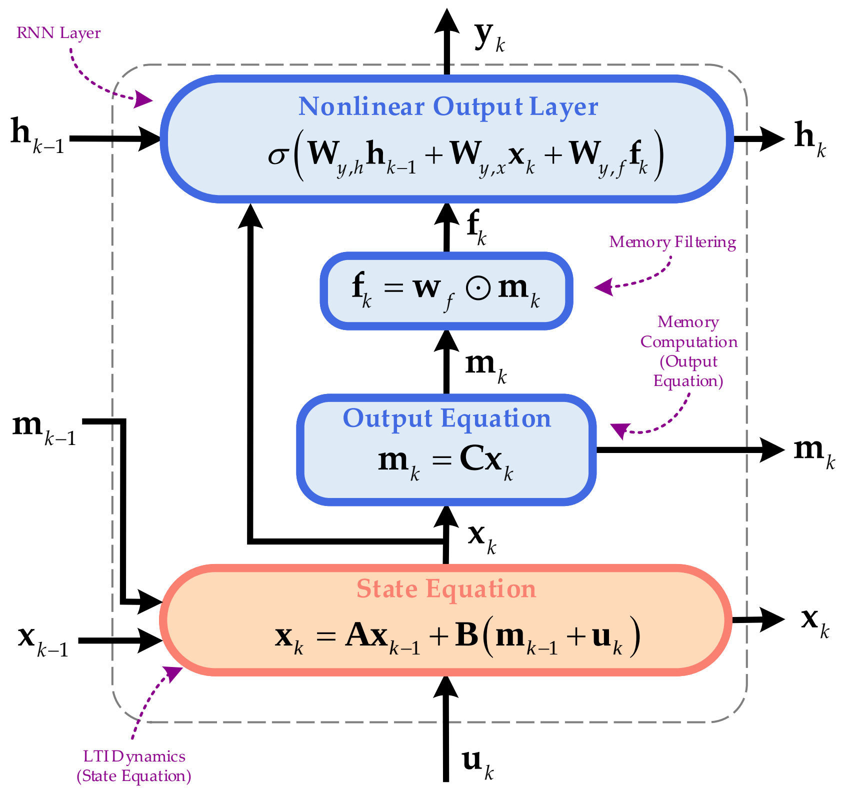

RNN Layer
$\mathbf{y}_k$
Nonlinear Output Layer
$\sigma\left(\mathbf{W}_{y,h}\mathbf{h}_{k-1} + \mathbf{W}_{y,x}\mathbf{x}_k + \mathbf{W}_{y,f}\mathbf{f}_k\right)$
$\mathbf{h}_{k-1}$
$\mathbf{h}_k$
$\mathbf{f}_k$
Memory Filtering
$\mathbf{f}_k = \mathbf{w}_f \odot \mathbf{m}_k$
$\mathbf{m}_k$
Memory Computation (Output Equation)
$\mathbf{m}_{k-1}$
Output Equation
$\mathbf{m}_k = \mathbf{C}\mathbf{x}_k$
$\mathbf{m}_k$
$\mathbf{x}_k$
State Equation
$\mathbf{x}_k = \mathbf{A}\mathbf{x}_{k-1} + \mathbf{B}\left(\mathbf{m}_{k-1} + \mathbf{u}_k\right)$
$\mathbf{x}_{k-1}$
$\mathbf{x}_k$
$\mathbf{u}_k$
LTI Dynamics (State Equation)

Figure 9.3: Detailed orthogonal Laguerre RNN architecture.

$\mathbf{m}_k$ through the weights $\mathbf{W}_{y,h} \in \mathbb{R}^{n_y \times n_y}$, $\mathbf{W}_{y,x} \in \mathbb{R}^{n_y \times n}$ and $\mathbf{W}_{y,f} \in \mathbb{R}^{n_y \times n_u}$. The logits are passed through a nonlinear activation function $\boldsymbol{\sigma}$ that counts with a bias term not shown in the diagram.

9.3.3.1 Laguerre Network

The Laguerre network uses a fixed pair $(\mathbf{A}, \mathbf{B})$ where $\mathbf{A} = \mathbf{A}_\ell$ and $\mathbf{B} = \mathbf{B}_\ell$. In other words, the RNN has to determine during training the coordinate matrix $\mathbf{C}$ that results most optimal to the task under consideration.

For this architecture, the order n (i.e., the number of Laguerre polynomials) is left as a design hyperparameter. Further hyperparameters are the scaling factor p (default, $p = 1$), the sampling time ΔT used for discretization (default, $\Delta T = 1$ s), and the sampling method (default, zero-order hold).

9.3.3.2 Ladder Network

The Ladder network employs discrete Laguerre networks with $a = 0$ (Eq. (9.14)) to introduce DT delay dynamics inside the RNN. For this reason, the matrices $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ are non-trainable (i.e., the coefficient matrix $\mathbf{C}$ is fixed). In fact, $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ correspond to a state-space realization for the Ladder system $\mathbf{G}(z) \in \mathbb{C}^{n_u \times n_u}$

$$\mathbf{G}(z) = \begin{bmatrix} \frac{1}{z^{m_1}} & 0 & \dots & 0 \\ 0 & \frac{1}{z^{m_2}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{z^{m_d}} \end{bmatrix} \tag{9.19}$$

where $m_1,\ m_2,\ \dots,\ m_{n_u-1},\ m_{n_u}$ are the delays of each channel, fulfilling $m_1 < m_2 < \cdots < m_{n_u}$. Note that $m_{n_u} = m_d$, the maximum delay, is specified by the user. Using this input, the algorithm to construct the Ladder network in transfer function form distributes the delays $m_1, m_2, \dots, m_{n_u-1}$ equally among all the input channels. For example, for a system with 100 inputs and a maximum delay of 100, $\mathbf{G}(z)$ has the form:

$$\mathbf{G}(z) = \begin{bmatrix} \frac{1}{z} & 0 & \dots & 0 \\ 0 & \frac{1}{z^2} & \vdots & 0 \\ \vdots & \dots & \ddots & \vdots \\ 0 & 0 & \vdots & \frac{1}{z^{100}} \end{bmatrix}. \tag{9.20}$$

Likewise, for a system with three inputs and $m_{n_u} = 100$, we have

$$\mathbf{G}(z) = \begin{bmatrix} \frac{1}{z} & 0 & 0 \\ 0 & \frac{1}{z^{50}} & 0 \\ 0 & 0 & \frac{1}{z^{100}} \end{bmatrix} \tag{9.21}$$

where the signal on the first, second and third channels will be delayed by one, 50 and 100 samples, respectively.

Once the matrix $\mathbf{G}(z)$ is constructed, a state-space representation is computed in controllable-canonical form. The resulting matrices $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ are set inside the RNN architecture (Fig. 9.3) to construct the Ladder variant. The implementation of the system $\mathbf{G}(z)$ inside the RNN structure relies on the library `python-control`[1].

[1]`https://github.com/python-control/python-control`.

Finally, it must be underlined that all experiments in Section 9.4 use the length of the TS (in number of samples) as the maximum delay m_d.

9.4 Experiments

The performance of the proposed architecture and their variants is evaluated experimentally in two benchmarks in which the RNN is challenged to learn dynamical behavior from data.

To generate the training data, the system of differential equations representing the continuous-time dynamics is integrated by an ODE solver instead of discretizing the CT dynamics and computing the outputs from the resulting DEs. Two different systems are studied: a pendulum and a fluid flow.

9.4.1 Time-Series Prediction from Dynamical Systems

For time-series prediction benchmark, the goal is to predict the behavior of a nonlinear system from a set of available measurements [127]. The experiments are performed for a pendulum with no friction, and a fluid flow system.

In contrast to the approach in [127], the problem is formulated in a supervised framework where the training data batches are generated by the keras functionality `TimeseriesGenerator`. Nevertheless, the splitting between training and testing data is maintained (35.29% for training, and 64.71% for testing). Results are reported for five different seeds.

The forecasting error is defined as the relative difference between the ground truth and the prediction at the final step:

$$e = \frac{\|\hat{y}_p - y\|_2}{\|y\|_2} \tag{9.22}$$

where $\hat{y}_p$ is the forecasted output and y is the ground truth. The experiments are performed in noiseless and noisy conditions (Gaussian noise) for two different systems: a pendulum and a fluid flow model.

Pendulum

The pendulum dynamics are represented as:

$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell}\sin\theta = 0 \tag{9.23}$$

with $g = 9.8$ and $\ell = 1$ [127]. In this case, the initial condition was changed so that the system will switch between linear ($\theta_0 = 0.8$) and nonlinear operation ($\theta_0 = 2.4$). Notice that this pendulum equation considers no friction. A state-space representation with $x_1 := \theta$ and $x_2 = \dot{\theta}$ is given by

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\frac{g}{\ell}\sin x_1. \end{aligned} \tag{9.24}$$

The data is generated by integrating the pendulum equations directly using a numerical method in the interval $[0, 170]$ s with $\Delta t = 0.1$ s.

Fluid Flow System

This benchmark represents a nonlinear fluid flow past a circular cylinder at Reynolds number 100 [128]. The system of ODEs employed to generate the data is expressed as

$$\begin{aligned} \dot{x}_1 &= \mu x_1 - \omega x_2 + A x_1 x_3 + u \\ \dot{x}_2 &= \omega x_1 + \mu x_2 + A x_2 x_3 \\ \dot{x}_3 &= -\lambda\left(x_3 - x_1^2 - x_2^2\right) \end{aligned} \tag{9.25}$$

with $\mu = 0.1$, $\omega = 1$, $A = -0.1$ and $\lambda = 10$. The initial condition is taken randomly. Data is generated for the range $[0, 2]$ s with $\Delta t = 0.001$ s.

9.4.2 System Identification

For this experiment, the learning problem aims at determining the state-space representation of a system from data as a regression task. This problem has been previously tackled with supervised [129] and unsupervised learning approaches [127].

Either way, the main difficulty of state-space identification is that it requires measuring the state of the system $\mathbf{x}$, which is typically not available. Usually, the state is estimated using measurements from the inputs and the outputs of the system which are usually measurable in practice. This information can be used to determine a black-box model of the system which does not reveal anything about its internal structure. This approach is widely used by dynamical systems practitioners, and has a direct connection to the black-box nature of an RNN.

For the experiments below, we consider that the input to the system and the initial state are available, which is a reasonable assumption in a practical environment. Furthermore, we take all states as outputs. By doing so, our benchmark is constrained to the dynamical system

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}\left(\mathbf{x}, u\right) \\ \mathbf{y} &= \mathbf{x} \end{aligned} \tag{9.26}$$

where u is the external input. This is a gray-box problem since we get access to the states which can describe the internal mechanisms of the system. Moreover, given the fact that the output of the RNN is the state of the system, the learning task can be understood as a state estimation problem. The discrete version of the dynamics in (9.26) is given by

$$\begin{aligned} \mathbf{x}_{k+1} &= \bar{\mathbf{f}}\left(\mathbf{x}_k, u_k\right) \\ \mathbf{y}_k &= \mathbf{x}_k. \end{aligned} \tag{9.27}$$

Eq. (9.27), despite its generality, is not into RNN form. To do so, a unit delay is applied to the state equation to get

$$\mathbf{x}_k = \bar{\mathbf{f}}\left(\mathbf{x}_{k-1}, u_{k-1}\right). \tag{9.28}$$

In this way, we see that the previous state and the previous output are required to update the state. This resembles more the RNN form $\mathbf{h}_k = \sigma\left(\mathbf{u}_k, \mathbf{h}_{k-1}\right)$. Considering sequences with n_{steps} time steps and a system with $n \in \mathbb{N}$ states, the state and the output of the discrete-time system are concatenated to construct the input $\mathbf{u}_{\text{RNN}} \in \mathbb{R}^{n_{\text{steps}} \times n}$ as follows[2]:

[2]For more details regarding the data generation and preprocessing for the system identification benchmark, see: https://github.com/AleksandarHaber/Machine-Learning-of-Dynamical-Systems-using-Recurrent-Neural-Networks.

$$\mathbf{u}_{\text{RNN}} = \begin{bmatrix} x_0^{(1)} & x_0^{(2)} & \dots & x_0^{(n)} \\ u_0 & 0 & \dots & 0 \\ u_1 & 0 & \vdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ u_{k-1} & 0 & \dots & 0 \end{bmatrix}. \tag{9.29}$$

Note that the first row of the input matrix is the transposed initial state vector $\mathbf{x}_0 = \begin{bmatrix} x_0^{(1)} & x_0^{(2)} & \dots x_0^{(n)} \end{bmatrix}^T$. The input values are placed in the first column since it is assumed that the RNN will deal only with a single output. This matrix unveils that the RNN sees both the state and the exogenous signal u as inputs.

After defining the input to the RNN, the output is generated by applying an input to the system and solving the underlying system of differential equations. For simplicity, we assume an arbitrary random initial state, although assuming the system to be initially relaxed (i.e., zero initial states) is a common practice. The input is taken to be a random signal to guarantee that most system modes are excited and can be detected in the output measurements.

In this way, training, testing, and validation data are produced by setting a different vector of initial conditions and a random input for each instance in the batch. Then, RNN training is formulated as a regression problem using mean squared error as a loss function, feeding the inputs $\mathbf{u}_{\text{RNN}}$ and computing the outputs $\mathbf{y} = \mathbf{x}$. For both experiments, 500 instances are generated, 10% of which is used for validation. Training is performed using k-fold cross-validation over 5-folds.

This experimentation is performed using two different nonlinear systems as benchmarks: a pendulum (2nd-order) and a fluid flow (3rd-order) dynamics. Performance is evaluated using averaged mean absolute error over five-folds.

Pendulum

The mathematical model of the pendulum is similar to that one in (9.24):

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -bx_2 - c\sin(x_1) + u \end{aligned} \tag{9.30}$$

with $b = 0.5$ and $c = 1.0$. The main difference is that viscous friction is considered. In this

Table 9.1: Prediction error at final step averaged over 5-fold for the dynamic system prediction experiment for five different seeds.

Model	Pendulum (no noise; linear)	Pendulum (noisy; linear)	Pendulum (no noise; nonlinear)	Pendulum (noisy; nonlinear)	Fluid Flow (no noise)	Fluid Flow (noisy)
Ladder	0.0150 ± 0.0050	**0.0116 ± 0.0018**	0.1819 ± 0.1041	0.1929 ± 0.1261	**0.5348 ± 0.0373**	**0.5135 ± 0.0626**
Laguerre	**0.0146 ± 0.0029**	0.0168 ± 0.0057	**0.1249 ± 0.0060**	**0.1257 ± 0.0059**	0.5166 ± 0.1338	0.7973 ± 0.5440
LMU	0.2261 ± 0.0986	0.1914 ± 0.0817	0.5132 ± 0.2465	0.5012 ± 0.1133	1.3952 ± 0.0013	1.3942 ± 0.0025
BRC	21.8499 ± 1.8473	23.1749 ± 1.8363	14.0479 ± 4.1699	18.0374 ± 1.7233	0.7843 ± 0.1754	0.6973 ± 0.1684
nBRC	2.1914 ± 0.9910	1.7015 ± 0.4043	2.6853 ± 1.5883	3.0102 ± 1.0030	0.5939 ± 0.0266	0.5656 ± 0.0481

model, x_1 corresponds to the angular displacement of the pendulum, and x_2 is the angular speed. u (input to the system) corresponds to the force applied to the pendulum which causes its movement by creating a torque.

Fluid Flow

The fluid flow model is the same as in Equation (9.31) with an input u affecting the first state:

$$\begin{aligned} \dot{x}_1 &= \mu x_1 - \omega x_2 + A x_1 x_3 + u \\ \dot{x}_2 &= \omega x_1 + \mu x_2 + A x_2 x_3 \\ \dot{x}_3 &= -\lambda \left(x_3 - x_1^2 - x_2^2 \right). \end{aligned} \tag{9.31}$$

9.4.3 Implementation

All experiments have been implemented in TensorFlow 1 using the keras API. Training settings were adapted from those implemented by [73] and [113]. Consequently, the training routine has three main callbacks: `ModelCheckpoint` to save only the best model during training, `EarlyStopping` to avoid setting by hand the number of epochs for each model, and `ReduceLROnPlateau` to adjust the learning rate automatically if the algorithm is stuck in a plateau of the loss function.

The implementation of the Ladder and Laguerre architectures rely on the `python-control` library. For the Laguerre network, the $(\mathbf{A}, \mathbf{B})$ matrices are constructed from the CT equations (9.12) and (9.11). Then, a CT state-space object is constructed and discretized using the `sample` function of the `StateSpace` class. For the Ladder network, the transfer function $\mathbf{G}(z)$ is constructed firstly using numerator and denominator convolutions taking advantage of the mathematical representation of transfer functions of DT delays. Then, $\mathbf{G}(z)$ is converted into a state-space realization by `tf2ss` which returns the $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ matrices used for the implementation of the RNN.

Experiments were executed on three different machines. The characteristics are shown in Table 9.2. All servers run on Ubuntu 18.04.3 LTS. The commands to dispatch the simulations are available in the GitHub[3] repository.

Table 9.2: Details of the testing equipment.

Server	Processor	RAM	GPU
LittleMan	AMD Ryzen Threadripper 3960X (24 cores @ 2.20 GHz)	128 GB	(2x) Quadro RTX 6000 (24 GB each)
Fatboy	AMD Epyc (64 Cores @ 2.70 GHz)	512 GB	(2x) Titan RTX (24 GB each)
DeepGrid	Intel Xeon E5-1650 (6 Cores @ 3.60 GHz)	256 GB	(4x) GeForce GTX 1080 (12 GB each)

9.5 Discussion

9.5.1 Experimental Results

Results for the time-series prediction and the dynamical systems experiments are reported in Tables 9.1 and 9.4. All results are reported as an average over 5-fold with the corresponding standard deviation. For the time-series prediction benchmark, we see that the proposed Ladder and Laguerre architectures show the best performance for both systems in noiseless and noisy conditions.

In Fig. 9.4 we can see how the prediction error remains constant for the Laguerre architecture as the prediction step increases, whereas for LMU and Ladder the error increases

[3]https://github.com/sergio-dorado/LaguerreRNNs.

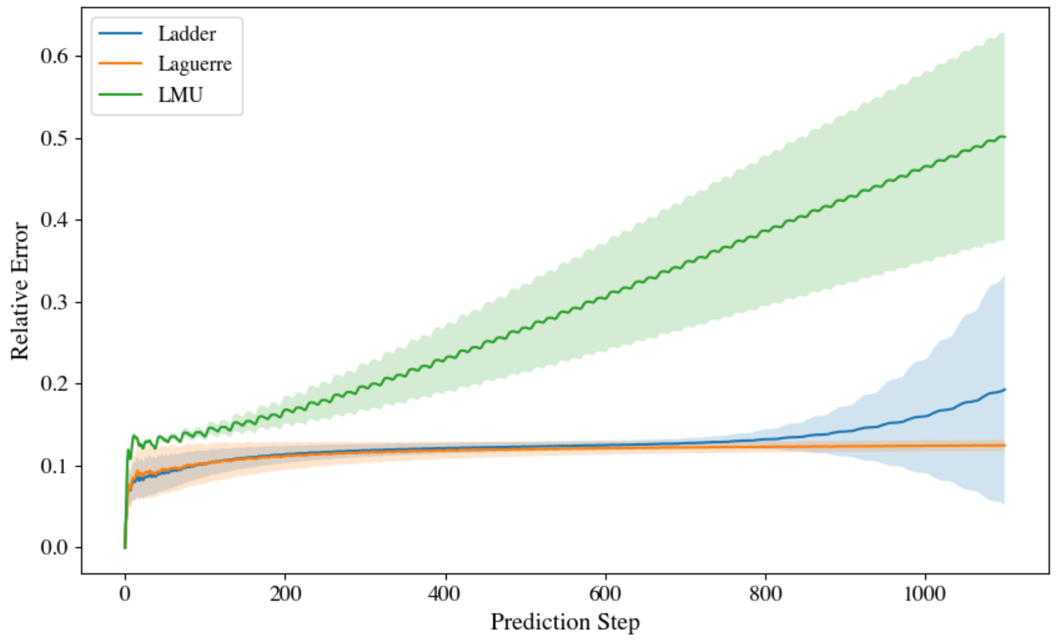


Figure 9.4: Relative prediction error for the noisy nonlinear pendulum experiment.

progressively accompanied by a larger variance. An important observation is that the Laguerre and Ladder network can adapt to nonlinear environments despite having a linear system construction.

Moreover, it was found empirically that the results were particularly sensitive to the optimizer choice. Results are reported using the `Fltr` algorithm.

As pointed out in [126], one of the virtues a dynamical system-based RNN such as an LMU is the reduction on the number of trainable compared to conventional architectures. For the case of Laguerre and Ladder networks, we must underline that the relationship between the number of the trainable parameters and the characteristics of the network is fully interpretable. Table 9.3 shows that the Ladder network balances the trade-off between number of trainable parameters and performance (the BRC architecture despite having the smallest number of trainable parameters shows the worst performance in almost all benchmarks). The Laguerre network counts with as many trainable parameters as LMUs.

It must be underlined that there exists a connection between the number of trainable parameters and the number of states of the network. Moreover, the larger the number of states, the more complex dynamics the memory policy can adapt. However, since the state is a part of the output layer, the network will need to train a larger number of weights for higher-order memory dynamics.

While controlling the states is also a characteristic of LMUs, their design requires a specification of the time shifting parameter to determine the orthogonality domain of the Legendre polynomials. This fact could be counterintuitive to a designer.

For the case of Laguerre units, only the number of states can be set directly by the designer. By doing so, all the advantages of controlling the number of states (and hence the number of trainable parameters) are preserved. Nevertheless, Laguerre layers require no further parameter specification in contrast to Legendre units.

On the other hand, the delay network needs to specify only the delay amount, from which the number of states is directly computed. This characteristic reduces the number of inputs while providing a straightforward interpretation to the designer.

Table 9.3: Number of parameters for the nonlinear pendulum experiment.

Model	**Trainable Parameters**
Ladder	**59,788**
Laguerre	100,792
LMU	100,536
BRC	4,802
nBRC	324,002

For the system identification experiments, all RNN architectures were capable of learning the system behavior with a small error. This is a consequence of the fact that a RNN is naturally a dynamic system and the parameters of the network can adapt its behavior to match the given data. In regards to the proposed architectures, we observe that the Laguerre network adjusts better to the nonlinear environment than the Ladder RNN.

Table 9.5: Characteristics of the UCR/psMNIST benchmarks.

Dataset	Task	Dimensions	Length	Instances	Number of Classes
psMNIST	Sequential Multi-class Classification	1	784	70000	10
Chlorine Concentration	Univariate Time-series Multi-class Classification	1	166	4307	3
Pen Digits	Multivariate Time-Series Multi-class Classification	2	8	10992	10
Phoneme Spectra	Multivariate Time-Series Multi-class Classification	11	217	6688	39
Wafer	Binary Time-Series Classification	1	152	7164	2

Table 9.4: MAE averaged over 5-fold for the system identification experiment.

Model	Pendulum	Fluid Flow
Ladder	**0.0133 ± 0.0012**	0.0206 ± 0.0095
Laguerre	0.0185 ± 0.0024	0.0037 ± 0.0001
LMU	0.0074 ± 0.0020	0.0066 ± 0.0003
BRC	0.0891 ± 0.0005	0.0040 ± 0.0009
nBRC	0.0067 ± 0.0065	**0.0014 ± 0.0005**

9.6 Further Discussion

9.6.1 Results on UCR and psMNIST datasets

The purpose of this section is to validate the performance of the proposed architectures on several standard datasets. Each of these was selected so that the testing conditions of the NN covered a broad range of scenarios where it could be deployed in practical applications. The characteristics of each dataset is presented in Table 9.5.

The psMNIST dataset has been employed ubiquitously by the Machine Learning community to assess the performance of novel RNN architectures (e.g., [113], [117]). Table 9.6 shows the experiment outcomes for the testing dataset in terms of the F1 scores. Notice that the Laguerre and Ladder architectures outperform conventional and recent proposals.

Table 9.6: F1 results for the psMNIST dataset.

Model	**Testing F1-Score**
Ladder	**0.9765**
Laguerre	0.9647
LMU	0.9580
BRC	0.6409
nBRC	0.9271
LMU	0.9157
LSTM	0.9092

Table 9.7 presents the results in terms of testing accuracy. The results of the models in blue were obtained in our experimental setup. The rest are reported in [117].

Table 9.7: Accuracy results for the psMNIST dataset (extended from [117]).

Model	**Testing Accuracy**
Ladder	**0.9767**
Laguerre	0.9650
LMU	0.9584
NRU	0.9538
EURNN	0.9450
nBRC	0.9280
SRU	0.9249
JANET	0.9194
GRU	0.9167
LSTM	0.9105
RNN.orth	0.8926
LSTM-chrono	0.8843
GORU	0.8700
RNN.id	0.8613
BRC	0.6485

For k-fold cross validation, with $k = 5$, we experienced some convergence problems to validate all benchmarking architectures. More specifically, the k-fold training algorithm did not converge for the BRC, LMU and nBRC models. Results for five folds on the F1 score are summarized in Table 9.8.

Table 9.8: F1 results for k-fold cross validation for the psMNIST dataset.

Model	Testing F1-Score
Ladder	**0.9952 ± 0.009**
Laguerre	0.9922 ± 0.014
LMU	0.9585 ± 0.029
LSTM	0.9674 ± 0.033

In regards to the UCR datasets, we see that our architectures rank among the best performing models for all experiments as shown in Table 9.9.

Table 9.9: Performance on the UCR datasets over 5-fold.

Model	Chlorine Concentration	Pen Digits	Phoneme Spectra	Wafer (AUROC)
Ladder	0.9995 ± 0.0012	**0.9979 ± 0.0036**	0.2206 ± 0.1208	**0.9999 ± 0.0000**
Laguerre	0.9987 ± 0.0012	0.9698 ± 0.0172	0.0398 ± 0.0097	0.9999 ± 0.0003
LMU	**0.9997 ± 0.0006**	0.9962 ± 0.0050	0.2760 ± 0.1721	0.9995 ± 0.0009
BRC	0.2981 ± 0.0122	0.9974 ± 0.0025	0.1725 ± 0.0554	0.9992 ± 0.0005
nBRC	0.3554 ± 0.0567	0.9977 ± 0.0025	**0.3598 ± 0.1197**	0.9986 ± 0.0012

Numerical Constraints on DT Laguerre Network Design

The purpose of this section is to illustrate some of the caveats of a direct discrete-time Laguerre network formulation. Let $\mathbf{l}_k \in \mathbb{R}^{N\times 1}$ be a vector expressing the value of the first N Laguerre functions at time k, $\mathbf{l}_k = \begin{bmatrix} \ell_k^{(1)} & \ell_k^{(2)} & \dots & \ell_k^{(N)} \end{bmatrix}^T$. The set of N Laguerre functions satisfy

$$\mathbf{l}_{k+1} = \mathbf{A}_\ell \mathbf{l}_k \tag{9.32}$$

with initial condition

$$\mathbf{l}_0 = \sqrt{\beta} \begin{bmatrix} 1 \\ -a \\ a^2 \\ -a^3 \\ \vdots \\ (-1)^{N-1} a^{N-1} \end{bmatrix} \tag{9.33}$$

The matrix $\mathbf{A}_\ell$ has some important characteristics, just as its continuous-time counterpart. First, it is lower-triangular

$$\mathbf{A}_\ell = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{bmatrix}. \tag{9.34}$$

Secondly, $\mathbf{A}_\ell$ is parametrized in terms of a and $\beta := (1 - a^2)$, and it has a Toeplitz structure. The first column of $\mathbf{A}_\ell$, written as $\mathbf{a}_\ell^1$, is:

$$\mathbf{a}_\ell^1 = \begin{bmatrix} a \\ \beta \\ -a\beta \\ a^2\beta \\ -a^3\beta \\ \vdots \\ (-a)^{N-2} \beta \end{bmatrix} \tag{9.35}$$

Moreover, $\mathbf{A}_\ell$ has a sparse characteristic. If $\mathbf{A}_\ell \in \mathbb{R}^{N \times N}$, the sparsity of $\mathbf{A}_\ell$ measured by the ratio of zero entries to the total number of elements is:

$$s_{\mathbf{A}_\ell} = \frac{\sum_{k=1}^{N-1} N - k}{N^2} = \frac{N-1}{2N} \tag{9.36}$$

The parameter a together with the order of the network N characterize completely the structure of the Laguerre representation. Theoretically, the higher the order of the network, the better the approximation to any unknown dynamics, including delays as happens with LMUs. However, a high-order Laguerre representation can show numerical problems since the $\mathbf{A}_\ell$ matrix can become too sparse.

This practical upper limit on the order of the approximation can be understood by analyzing the computation of the kth entry of $\mathbf{a}_\ell^1$ for $k \geq 3$:

$$\left[\mathbf{a}_\ell^1\right]_k = (-a)^{k-2}(1-a^2)$$

The magnitude of this expression is bounded by $|a|^k$. Since $0 \leq a < 1$ for stability, the kth entry of $\mathbf{a}_\ell^1$ will tend to zero for large k (Fig. 9.5. This increases the sparsity of $\mathbf{A}$, but, in the limit, will turn the matrix ill-conditioned. Despite this, an upper bound on N reduces the number of trainable parameters since it defines the number of states in Eq. (9.5). A conclusion from this analysis is that the use of a DT Laguerre representation will imply a reduced number of learned parameters in order to preserve numerical stability.

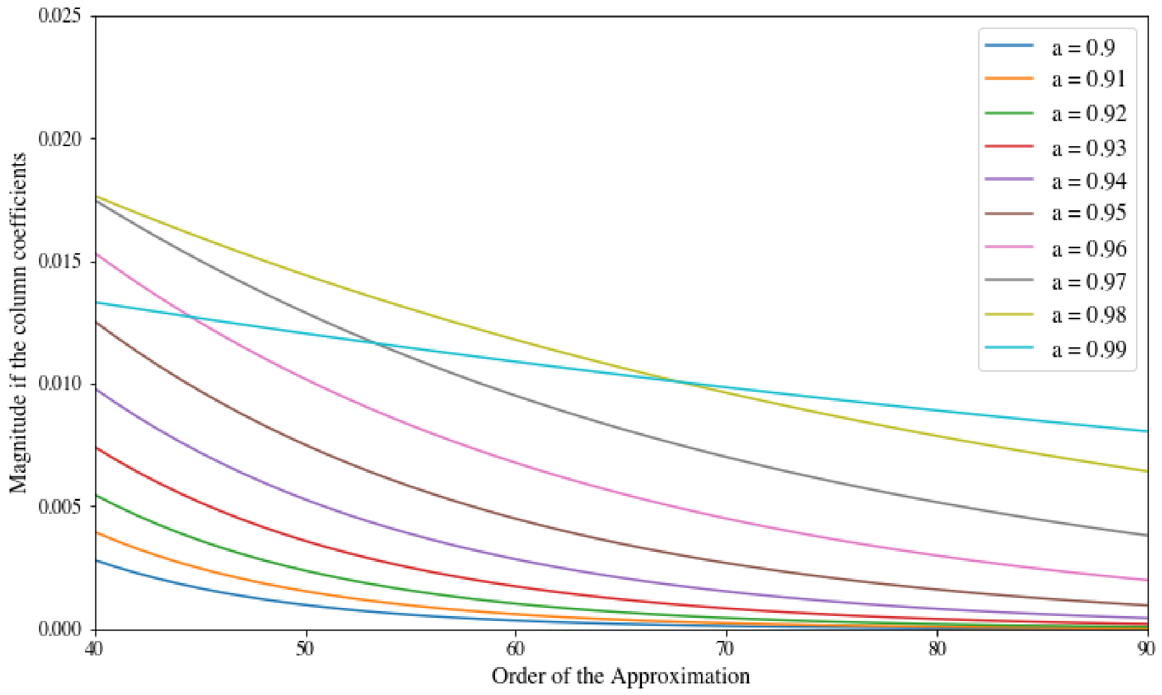


Figure 9.5: Magnitude of the non-zero entries of the Laguerre matrix as a function of the approximation order for several values of a.

It is possible to use float precision to exploit Laguerre functions at the expense of more memory usage. By setting the a value close to the stability boundary, the numerical condition of the $\mathbf{A}_\ell$ matrix for the discrete-time design improves. Fig. 9.6 depicts how increasing the order of the network leads to more ill-conditioned matrices $\mathbf{A}_\ell$. However, by increasing the value of a close to 1, the condition number can be improved for any Laguerre order of practical interest. Here, the numerical precision is fundamental: if a is taken as larger than 1, the dynamics in Eq. (9.5) will be unstable and the memory values will become unbounded after a few iterations.

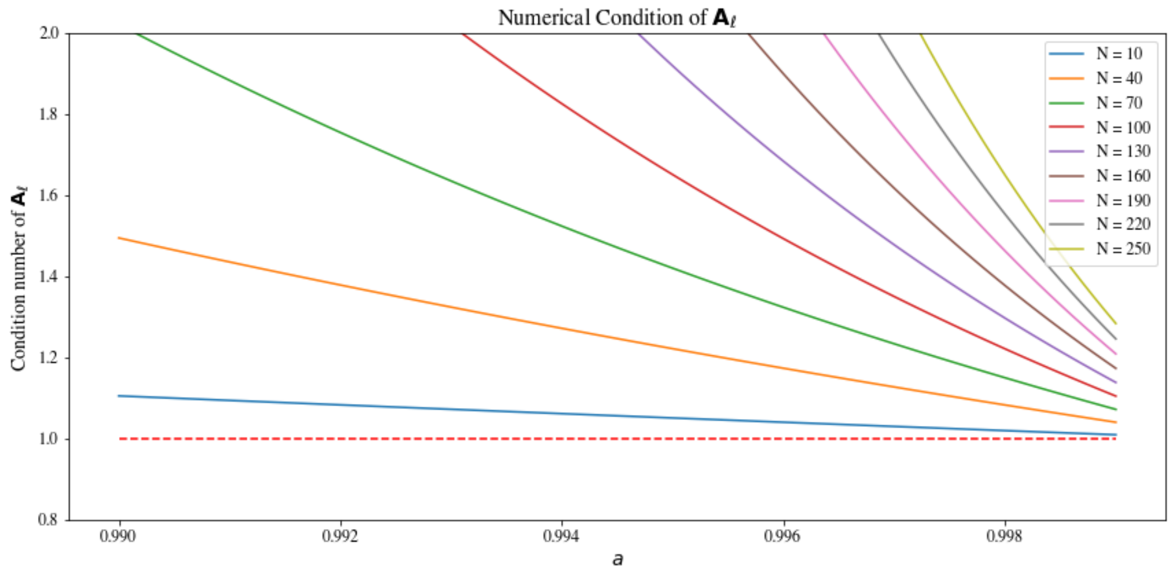


Figure 9.6: Condition number of the matrix $\mathbf{A}_\ell$ as a function of the parameter a.

9.7 Conclusions

This work has introduced two novel RNN architectures inspired by DT dynamical systems: the Laguerre and the Ladder network. In both cases, the memory update policy of the RNN is constrained by the state equation of a DTLTI system, which is fully parametrized in terms of the matrices $(\mathbf{A}, \mathbf{B})$. For the Laguerre network, $(\mathbf{A}, \mathbf{B})$ are obtained after discretizing the CT Laguerre polynomials. This set of functions has the desirable property of orthonormality based on which they can be used to represent any stable DTLTI system through the coordinate matrix $\mathbf{C}$. $\mathbf{C}$ is learned during training by the Laguerre RNN so that the RNN steers the dynamical behavior of the memory towards the optimal LTI system according to the cost function.

For the Ladder network, $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ are fixed so that the memory exhibits a delay behavior from input to output. This design is carried out directly in the discrete-time domain where the convenience of the mathematical representation of discrete-time delays in frequency domain are exploited to embed such system inside the RNN representation. Thanks to the direct connection of both architectures to LTI system theory, the RNN behavior is explainable from the dynamical systems perspective: the inclusion of an LTI basis (Laguerre polynomials) enhances the expressibility of the network, and the delay input-output behavior guarantees dynamically that the memory flow is preserved through time.

The performance of the networks was evaluating in a forecasting benchmark where the RNN is subjected to predict dynamical systems data. The experimental result suggests that both architectures perform better than other dynamics-based RNN proposals which have received attention recently. These promising empirical results suggest potential for deployment in industrial applications as a building block for AI-based solutions.

CHAPTER 10
CONCLUSIONS

In the context of 21st-century power systems analysis, this document presented a methodology to generate synthetic data via phasor time-domain simulations in Modelica. Also, we profit from the potential of data to develop solutions to assist power system operators in tasks such as small-signal analysis and forced oscillation detection.

In Chapter 2, we introduced a benchmark analysis to underline the advantages and disadvantages of different Modelica IDEss from the point of view of simulation performance. This study helps users make an educated tool and solver selection according to their particular needs. In Chapter 3, we dive deeper into the importance of power flow computations for power system dynamic simulations. We employed GridCal, an open-source Python library, to perform power flow computations. The obtained power flow results are close to solver tolerance to those of PSS®E, the industry standard for dynamical studies in power systems. In addition, we introduced a novel data structure taking advantage of the Modelica `Records` class to handle power flow variables. Researchers in other domains can benefit from such a hierarchical, object-oriented approach for efficient parameter management in the Modelica environments.

Chapter 4 introduced a method for contingency generation based on two-stage Monte Carlo sampling. Also, we presented the first application of synthetic data in developing a small-signal stability classifier using "conventional" or "classical" ML techniques.

Chapter 5 described in detail one of the most relevant contributions of this thesis: `ModelicaGridData`. This Python-based software tool utilizes load profiles obtained from the NYISO to generate different initial conditions. The resulting power flows are employed for simulating highly non-linear power system models under contingency scenarios. The tool also counts with a data post-processing routine where the user can extract different signals from the simulation data.

Afterwards, applications of data-driven techniques in power grids were discussed. Chapter 6 introduced a method for time series-based small-signal stability assessment based on CNNs. In contrast to conventional (signal processing-based or system identification techniques), we showed that the CNNs show a remarkable computational efficiency. This characteristic of DL methods invited us to explore the potential for real-time applications, which is the subject of Chapters 7 and 8. The former described a low-cost hardware platform that can validate the performance of ML solution deployed in IoT devices for inference at the edge. Chapter 9 presented an application of DL techniques for forced oscillation detection at the edge, showing excellent performance when subjected to ambient and synthetic data.

Finally, Chapter 9 introduced a novel RNN architecture inspired by DTLTI systems. The Laguerre network employs a Laguerre polynomial representation as a basis and learns the coordinates of the stable LTI system that best suits the underlying task. The Ladder network embeds the dynamics of DT delays to increase the long-term memory retention of the RNN. Both architectures show state-of-the-art performance in several benchmarks for classifying and forecasting time series data. This kind of physics-aware solution is fundamental for conceiving algorithms to help the power system engineers operate the sustainable and greener electrical networks of the 21st-century.

REFERENCES

[1] F. Milano, F. Dörfler, G. Hug, D. J. Hill, and G. Verbič, "Foundations and Challenges of Low-Inertia Systems (Invited Paper)," in *2018 Power Syst. Comp. Conf. (PSCC)*, Jun. 2018.

[2] J. A. Taylor, *Convex Optimization of Power Systems*. Cambridge University Press, 2015.

[3] A. Navon *et al.*, "Applications of Game Theory to Design and Operation of Modern Power Systems: A Comprehensive Review," *Energies*, vol. 13, no. 15, Aug. 2020.

[4] E. O. Arwa and K. A. Folly, "Reinforcement Learning Techniques for Optimal Power Control in Grid-Connected Microgrids: A Comprehensive Review," *IEEE Access*, vol. 8, 2020.

[5] G. Laera *et al.*, "Guidelines and Use Cases for Power Systems Dynamic Modeling and Model Verification using Modelica," in *Proc. of the American Modelica Conf. 2022, Houston, Texas, USA*, Oct. 2022.

[6] S. A. Dorado-Rojas, M. Navarro Catalán, M. de Castro Fernandes, and L. Vanfretti, "Performance Benchmark of Modelica Time-domain Power System Automated Simulations using Python," in *Proc. of the American Modelica Conf. 2020, Boulder, Colorado, USA*, Mar. 2020.

[7] S. A. Dorado-Rojas, G. Laera, M. de Castro Fernandes, T. Bogodorova, and L. Vanfretti, "Power Flow Record Structures to Initialize OpenIPSL Phasor Time-Domain Simulations with Python," in *Proc. of 14th Modelica Conf. 2021*, Sep. 2021.

[8] S. A. Dorado-Rojas, M. de Castro Fernandes, and L. Vanfretti, "Synthetic Training Data Generation for ML-based Small-Signal Stability Assessment," in *2020 IEEE Int. Conf. on Comm., Contr., and Comp. Tech. for Smart Grids (SmartGridComm)*, Nov. 2020.

[9] S. A. Dorado-Rojas, F. Fachini, T. Bogodorova, G. Laera, M. de Castro Fernandes, and L. Vanfretti, "ModelicaGridData: Massive Power System Simulation Data Generation and Labeling Tool using Modelica and Python," *SoftwareX*, 2022.

[10] S. A. Dorado-Rojas, T. Bogodorova, and L. Vanfretti, "Time Series-Based Small-Signal Stability Assessment using Deep Learning," in *2021 North American Power Symp. (NAPS)*, Nov. 2021.

[11] S. A. Dorado-Rojas, S. Xu, L. Vanfretti, G. Olvera, M. I. I. Ayachi, and S. Ahmed, "Low-Cost Hardware Platform for Testing ML-Based Edge Power Grid Oscillation Detectors," in *2022 10th Workshop on Mod. and Sim. of Cyber-Physical Energy Syst. (MSCPES)*, May 2022.

[12] S. A. Dorado-Rojas, S. Xu, L. Vanfretti, M Ilies, I Ayachi, and S. Ahmed, "ML-based Edge Application for Detection of Forced Oscillations in Power Grids," in *2022 IEEE Power & Energy Soc. General Meeting (PESGM)*, Jul. 2022.

[13] S. A. Dorado-Rojas, B. Vinzamuri, and L. Vanfretti, "Orthogonal Laguerre Recurrent Neural Networks," in *Mach. Learn. and the Phys. Sci. Workshop at the 34th Conf. on Neural Info. Proc. Syst. (NeurIPS)*, 2020.

[14] F. Milano, *Power System Modelling and Scripting.* Springer Berlin Heidelberg, Aug. 2010.

[15] M. Baudette, M. Castro, T. Rabuzin, J. Lavenius, T. Bogodorova, and L. Vanfretti, "OpenIPSL: Open-Instance Power System Library — Update 1.5 to "iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations"," *SoftwareX*, vol. 7, Jan. 2018.

[16] P. Fritzson *et al.*, "OpenModelica - A free open-source environment for system modeling, simulation, and teaching," in *2006 IEEE Conf. on Computer Aided Contr. Syst. Des.*, Oct. 2006.

[17] W. Braun, F. Casella, and B. Bachmann, "Solving large-scale Modelica models: new approaches and experimental results using OpenModelica," in *Proc. of the 12th Int. Modelica Conf., Prague, Czech Republic*, Jul. 2017.

[18] E. Henningsson, H. Olsson, and L. Vanfretti, "DAE solvers for large-scale hybrid models," in *Proc. of the 13th Int. Modelica Conf., Regensburg, Germany*, Mar. 2019.

[19] B. Lie *et al.*, "API for Accessing OpenModelica Models from Python," in *Proc. of The 9th EUROSIM Congr. on Mod. and Sim., EUROSIM 2016, The 57th SIMS Conf. on Sim. and Mod. SIMS 2016*, Dec. 2018.

[20] J. L. Devore and K. N. Berk, *Modern Mathematical Statistics with Applications.* Springer New York, 2012.

[21] F. J. Gómez, M. Aguilera Chaves, L. Vanfretti, and S. H. Olsen, "Multi-Domain Semantic Information and Physical Behavior Modeling of Power Systems and Gas Turbines Expanding the Common Information Model," *IEEE Access*, vol. 6, 2018.

[22] S Boersma *et al.*, "Enhanced Power System Damping Estimation via Optimal Probing Signal Design," in *2020 22nd European Conf. on Power Electron. and App. (EPE'20 ECCE Europe)*, Sep. 2020.

[23] M. Podlaski, M. de Castro Fernandes, J. Pesente, and L. Vanfretti, "Parameter Estimation of User-Defined Control System Models for Itaipú Power Plant using Modelica and OpenIPSL," in *Proc. of the American Modelica Conf. 2020, Boulder, Colorado, USA*, Nov. 2020.

[24] K. Noháč, L. Raková, V. Mužik, and K. Máslo, "Open Source Platforms for Dynamic Stability Assessment," in *2019 20th Int. Sci. Conf. on Electr. Power Eng. (EPE)*, May 2019.

[25] D. Gusain, M. Cvetković, and P. Palensky, "Energy Flexibility Analysis using FMUWorld," in *2019 IEEE Milano Power Tech*, Jun. 2019.

[26] D. Winkler, "Analysing the stability of an Islanded hydro-electric power system," Feb. 2019.

[27] Y. Qin, M. Korkali, P. Top, and L. Min, "A JModelica.org Library for Power Grid Dynamic Simulation with Wind Turbine Control," in *2019 IEEE Power & Energy Soc. General Meeting (PESGM)*, IEEE, Aug. 2019.

[28] K. Pan, D. Gusain, and P. Palensky, "Modelica-Supported Attack Impact Evaluation in Cyber Physical Energy System," in *2019 IEEE 19th Int. Symp. on High Assurance Syst. Eng. (HASE)*, IEEE, Jan. 2019.

[29] J. C. Gonzalez-Torres, J Mermet-Guyennet, S Silvant, and A Benchaib, "Power system stability enhancement via VSC-HVDC control using remote signals: application on the Nordic 44-bus test system," in *15th IET Int. Conf. on AC and DC Power Trans. (ACDC 2019)*, IET, 2019.

[30] J. Müller, M. Baudette, D. Arnold, and M. Sankur, "A modelica library for continuous and discrete extremum seeking for static and dynamic systems," in *Proc. of the American Modelica Conf. 2020, Boulder, Colorado*, Nov. 2020.

[31] L Vanfretti, T Rabuzin, M Baudette, and M Murad, "iTesla Power Systems Library (iPSL): A Modelica library for phasor time-domain simulations," en, *SoftwareX*, vol. 5, 2016.

[32] D. Winkler, "T power system modelling in modelica - comparing open-source library options," in *Proc. of the 58th Conf. on Sim. and Mod. (SIMS 58) Reykjavik, Iceland*, Sep. 2017.

[33] B. Stott, "Review of load-flow calculation methods," *Proc. of the IEEE*, vol. 62, no. 7, 1974.

[34] F. Milano, "Continuous newton's method for power flow analysis," *IEEE Trans. on Power Syst.*, vol. 24, no. 1, Feb. 2009.

[35] J. B. Ward and H. W. Hale, "Digital computer solution of power-flow problems [includes discussion]," *Trans. of the American Institute of Electr. Engineers. Part III: Power App. and Syst.*, vol. 75, no. 3, 1956.

[36] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proc. of the IEEE*, vol. 55, no. 11, 1967.

[37] L. Vanfretti, T. Bogodorova, and S. A. Dorado-Rojas, *Power System Machine Learning Applications: From Physics-Informed Learning for Decision Support to Inference at the Edge for Control*, `https://github.com/ALSETLab/Tutorial_SGC_2020`, Accessed: 2022-7-14.

[38] N. Vyakaranam Bharat Samaan *et al.*, "Dynamic Contingency Analysis Tool 2.0 User Manual with Test System Examples," Pacific Northwest National Laboratory, Tech. Rep., 2019.

[39] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow*, 2nd ed. O'Reilly, 2019.

[40] S. A. R. Konakalla and R. A. de Callafon, "Feature Based Grid Event Classification from Synchrophasor Data," *Procedia Comp. Sci.*, vol. 108, 2017.

[41] C. Zheng, V. Malbasa, and M. Kezunovic, “Regression tree for stability margin prediction using synchrophasor measurements,” *IEEE Trans. on Power Syst.*, vol. 28, no. 2, May 2013.

[42] J. D. Pinzón and D. G. Colomé, “Real-time multi-state classification of short-term voltage stability based on multivariate time series machine learning,” *Int. J. of Electr. Power & Energy Syst.*, vol. 108, Jun. 2019.

[43] H.-Y. Su and T.-Y. Liu, “Enhanced-Online-Random-Forest Model for Static Voltage Stability Assessment Using Wide Area Measurements,” *IEEE Trans. on Power Syst.*, vol. 33, no. 6, Nov. 2018.

[44] W. D. Oliveira, J. P. A. Vieira, U. H. Bezerra, D. A. Martins, and B. d. G. Rodrigues, “Power system security assessment for multiple contingencies using multiway decision tree,” *Electr. Power Syst. Res.*, vol. 148, Jul. 2017.

[45] I. Konstantelos *et al.*, “Implementation of a Massively Parallel Dynamic Security Assessment Platform for Large-Scale Grids,” in *2018 IEEE Power & Energy Soc. General Meeting (PESGM)*, IEEE, Aug. 2018.

[46] W. Peres, E. J. de Oliveira, J. A. Passos Filho, and I. C. da Silva Junior, “Coordinated tuning of power system stabilizers using bio-inspired algorithms,” *Int. J. of Electr. Power & Energy Syst.*, vol. 64, Jan. 2015.

[47] D. Chitara, K. R. Niazi, A. Swarnkar, and N. Gupta, “Cuckoo Search Optimization Algorithm for Designing of a Multimachine Power System Stabilizer,” *IEEE Trans. on Ind. Appl.*, vol. 54, no. 4, Jul. 2018.

[48] M. J. Rana, M. S. Shahriar, and M. Shafiullah, “Levenberg–Marquardt neural network to estimate UPFC-coordinated PSS parameters to enhance power system stability,” *Neural Comp. & Appl.*, vol. 31, no. 4, Apr. 2019.

[49] F. R. S. Sevilla and L. Vanfretti, “A small-signal stability index for power system dynamic impact assessment using time-domain simulations,” in *2014 IEEE PES General Meeting (PESGM)*, Jul. 2014.

[50] J. H. Chow and J. J. Sanchez-Gasca, *Power System Modeling, Computation, and Control*, en. John Wiley & Sons, Jan. 2020.

[51] Dassault Systèmes, *Dymola User Manual.* Dassault Systèmes, 2018.

[52] Z. Zhuo, E. Du, N. Zhang, C. Kang, Q. Xia, and Z. Wang, “Incorporating Massive Scenarios in Transmission Expansion Planning with High Renewable Energy Penetration,” *IEEE Trans. on Power Syst.*, 2019.

[53] C.-T. Chen, *Analog and Digital Control System Design: Transfer-Function, State-Space, and Algebraic Methods.* Oxford University Press, Feb. 2006.

[54] J. F. Kavanagh, “Resistance as motivation for innovation: Open source software,” *Comm. of the Ass. for Info. Syst.*, vol. 13, 2004.

[55] WECC, “Western interconnection data sharing agreement,” Tech. Rep., 2018. [Online]. Available: `https://www.spp.org/Documents/58886/WIDSA\%20CLEAN_101618.docx`.

[56] I. Idehen, W. Jang, and T. Overbye, “PMU Data Feature Considerations for Realistic, Synthetic Data Generation,” in *2019 North American Power Symp. (NAPS)*, Oct. 2019.

[57] H. Li, J. H. Yeo, A. L. Bornsheuer, and T. J. Overbye, “The Creation and Validation of Load Time Series for Synthetic Electric Power Systems,” *IEEE Trans. on Power Syst.*, vol. 36, no. 2, Mar. 2021.

[58] P. H. Larsen, M. Lawson, K. H. LaCommare, and J. H. Eto, “Severe weather, utility spending, and the long-term reliability of the U.S. power system,” *Energy*, vol. 198, May 2020.

[59] S. Soltan, A. Loh, and G. Zussman, “A Learning-Based Method for Generating Synthetic Power Grids,” *IEEE Syst. J.*, vol. 13, no. 1, Mar. 2019.

[60] M. Khodayar, J. Wang, and Z. Wang, “Deep Generative Graph Distribution Learning for Synthetic Power Grids,” Jan. 2019.

[61] T. Bogodorova, D. Osipov, and L. Vanfretti, “Automated Design of Realistic Contingencies for Big Data Generation,” *IEEE Trans. on Power Syst.*, vol. 35, no. 6, Nov. 2020.

[62] *NYISO: New York Independent System Operator*, en, `https://www.nyiso.com/`, Accessed: 2022-6-17.

[63] *GridCal*, en, `https://www.gridcal.org/`, Accessed: 2022-6-17.

[64] M. de Castro and L. Vanfretti, “Multi Time-Scale Modeling of a STATCOM and Power Grid for Stability Studies using Modelica,” in *2022 Open Source Mod. and Sim. of Energy Syst. (OSMSES)*, Apr. 2022.

[65] J. C. Gonzalez–Torres, R. Mourouvin, K. Shinoda, A. Zama, and A. Benchaib, “A simplified approach to model grid-forming controlled MMCs in power system stability studies,” in *2021 IEEE Power & Energy Soc. Innov. Smart Grid Tech. Europe (ISGT Europe)*, Oct. 2021.

[66] F Fachini, L Vanfretti, M de Castro, and others, “Modeling and Validation of Renewable Energy Sources in the OpenIPSL Modelica Library,” *IECON 2021–47th*, 2021.

[67] F. R. S. Sevilla and L. Vanfretti, “Static stability indexes for classification of power system time-domain simulations,” in *2015 IEEE Power & Energy Soc. Innov. Smart Grid Tech. Conf. (ISGT)*, IEEE, Feb. 2015.

[68] *ModelicaGridData*, `https://www.youtube.com/watch?v=9KvV3BtDZuk&list=PLQSmZRKvTZtH940Oaw5yybX3RQaHNXpmZ`, Accessed: 2022-7-01.

[69] A. Collette, *Python and HDF5: Unlocking Scientific Data.* O’Reilly Media, Inc., Oct. 2013.

[70] Jared Council, *Data Challenges Are Halting AI Projects, IBM Executive Says*, `https://www.wsj.com/articles/data-challenges-are-halting-ai-projects-ibm-executive-says-11559035800`, Accessed: 2021-9-27, May 2019.

[71] J. F. Hauer, C. J. Demeure, and L. L. Scharf, "Initial results in prony analysis of power system response signals," *IEEE Trans. on Power Syst.*, vol. 5, no. 1, pp. 80–89, 1990.

[72] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016.

[73] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: a review," Sep. 2018.

[74] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *2017 Int. Joint Conf. on Neural Netw. (IJCNN)*, IEEE, 2017, pp. 1578–1585.

[75] B. Zhao, H. Lu, S. Chen, J. Liu, and D. Wu, "Convolutional neural networks for time series classification," *J. of Syst. Eng. and Electron.*, vol. 28, no. 1, pp. 162–169, 2017.

[76] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. of the IEEE Conf. on Comp. Vis. and Pattern Recognit.*, 2015, pp. 1–9.

[77] Y. Zheng, Q. Liu, E. Chen, Y. Ge, and J. L. Zhao, "Exploiting multi-channels deep convolutional neural networks for multivariate time series classification," *Front. of Comp. Sci.*, vol. 10, no. 1, pp. 96–112, 2016.

[78] J. Sanchez-Gasca and J. Chow, "Performance comparison of three identification methods for the analysis of electromechanical oscillations," *IEEE Trans. on Power Syst.*, vol. 14, no. 3, pp. 995–1002, 1999.

[79] M. Ghorbaniparvar, "Survey on forced oscillations in power system," *J. of Mod. Power Syst. and Clean Energy*, vol. 5, no. 5, Sep. 2017.

[80] M. A. Magdy and F Coowar, "Frequency domain analysis of power system forced oscillations," *IEE Proc. C (Gen., Transm., and Dist.)*, vol. 137, no. 4, Jul. 1990.

[81] G. Rogers, *Power System Oscillations*, en. Springer Science & Business Media, Dec. 2012.

[82] L. Vanfretti, S. Bengtsson, V. S. Perić, and J. O. Gjerde, "Spectral estimation of low-frequency oscillations in the Nordic grid using ambient synchrophasor data under the presence of forced oscillations," in *2013 IEEE Grenoble Conf.*, Jun. 2013.

[83] L. Vanfretti, M. Baudette, J. L. Domínguez-García, A. White, M. S. Almas, and J. O. Gjerdeóy, "A PMU-based fast real-time sub-synchronous oscillation detection application," in *2015 IEEE 15th Int. Conf. on Env. and Electr. Eng. (EEEIC)*, Jun. 2015.

[84] J. L. Domínguez-García *et al.*, "Validation experiment design of a PMU-based application for detection of sub-synchronous oscillations," in *2015 IEEE 15th Int. Conf. on Env. and Electr. Eng. (EEEIC)*, Jun. 2015.

[85] H. Cho, S. Oh, S. Nam, and B. Lee, "Non-linear dynamics based sub-synchronous resonance index by using power system measurement data," *IET Gen., Transm. and Dist.*, vol. 12, no. 17, 2018.

[86] H. Khalilinia and V. Venkatasubramanian, "Subsynchronous Resonance Monitoring Using Ambient High Speed Sensor Data," *IEEE Trans. on Power Syst.*, vol. 31, no. 2, Mar. 2016.

[87] A. Ghasempour, “Internet of Things in Smart Grid: Architecture, Applications, Services, Key Technologies, and Challenges,” *Inventions*, vol. 4, no. 1, Mar. 2019.

[88] A. Ghasemkhani, A. Darvishi, I. Niazazari, A. Darvishi, H. Livani, and L. Yang, “DeepGrid: Robust Deep Reinforcement Learning-based Contingency Management,” in *2020 IEEE Power & Energy Soc. Innov. Smart Grid Tech. Conf. (ISGT)*, Feb. 2020.

[89] M. S. Almas and L Vanfretti, *Experimental performance assessment of a generator’s excitation control system using real-time hardware-in-the-loop simulation*, 2014.

[90] S. Y. Gelbal, S. Tamilarasan, M. R. Cantas, L. Güvenc, and B. Aksun-Güvenc, “A connected and autonomous vehicle hardware-in-the-loop simulator for developing automated driving algorithms,” in *2017 IEEE Int. Conf. on Syst., Man, and Cybern. (SMC)*, Oct. 2017.

[91] L. Vanfretti, M. Baudette, I. Al-Khatib, M. S. Almas, and J. O. Gjerde, “Testing and validation of a fast real-time oscillation detection PMU-based application for wind-farm monitoring,” in *2013 First Int. Black Sea Conf. on Comm. and Netw. (BlackSeaCom)*, Jul. 2013.

[92] A Ghasempour and J Lou, “Advanced metering infrastructure in smart grid: Requirements, challenges, architectures, technologies, and optimizations,” in *Smart Grids: Emerg. Tech., Chall. and Fut. Dir.* Nova Science Publishers Hauppauge, 2017.

[93] P. B. Reddy and I. A. Hiskens, “Limit-induced stable limit cycles in power systems,” in *2005 IEEE Russia Power Tech*, Jun. 2005.

[94] W. Xuanyin, L. Xiaoxiao, and L. Fushang, “Analysis on oscillation in electro-hydraulic regulating system of steam turbine and fault diagnosis based on PSOBP,” *Expert Syst. with Appl.*, vol. 37, no. 5, May 2010.

[95] R. Xie and D. Trudnowski, “Distinguishing features of natural and forced oscillations,” in *2015 IEEE Power & Energy Soc. General Meeting (PESGM)*, Jul. 2015.

[96] V. Jain, S. T. Nagarajan, and R. Garg, “Study of Forced Oscillations in Two Area Power System,” in *2018 2nd IEEE Int. Conf. on Power Electron., Intell. Contr. and Energy Syst. (ICPEICES)*, Oct. 2018.

[97] J. Follum and J. W. Pierre, “Detection of Periodic Forced Oscillations in Power Systems,” *IEEE Trans. on Power Syst.*, vol. 31, no. 3, May 2016.

[98] F. Ghorbaniparvar and H. Sangrody, “PMU application for locating the source of forced oscillations in smart grids,” in *2018 IEEE Power and Energy Conf. at Illinois (PECI)*, Feb. 2018.

[99] D. J. Trudnowski and R. Guttromson, “A Strategy for Forced Oscillation Suppression,” *IEEE Trans. on Power Syst.*, vol. 35, no. 6, Nov. 2020.

[100] W. Hu, J. Liang, Y. Jin, F. Wu, X. Wang, and E. Chen, “Online Evaluation Method for Low Frequency Oscillation Stability in a Power System Based on Improved XGboost,” en, *Energies*, vol. 11, no. 11, Nov. 2018.

[101] D. N. Sidorov, Y. A. Grishin, and V. Šmidl, "On-line detection of inter-area oscillations using forgetting approach for power systems monitoring," in *2010 The 2nd Int. Conf. on Comp. and Autom. Eng. (ICCAE)*, vol. 3, Feb. 2010.

[102] Analytic Methods for Power Syst. (AMPS) Committee Transient Analysis and Simulations Subcommittee (TASS) Wind SSO Task Force, "Wind Energy Systems Sub-Synchronous Oscillations: Events and Modeling," IEEE Power & Energy Soc., Tech. Rep. PES-TR80, Jul. 2020.

[103] J. Liu, W. Yao, J. Wen, H. He, and X. Zheng, "Active Power Oscillation Property Classification of Electric Power Systems Based on SVM," en, *J. of Appl. Math.*, vol. 2014, May 2014.

[104] M.-I. Ayachi, L. Vanfretti, and S. Ahmed, "A PMU-Based Machine Learning Application for Fast Detection of Forced Oscillations from Wind Farms," Dec. 2020.

[105] *Speed up TensorFlow Inference on GPUs with TensorRT*, `https://blog.tensorflow.org/2018/04/speed-up-tensorflow-inference-on-gpus-tensorRT.html`, Accessed: 2021-10-26.

[106] A. Bokovoy, K. Muravyev, and K. Yakovlev, "Real-time Vision-based Depth Reconstruction with NVidia Jetson," in *2019 European Conf. on Mobile Robots (ECMR)*, Sep. 2019.

[107] M. Goyal, N. D. Reeves, S. Rajbhandari, and M. H. Yap, "Robust Methods for Real-Time Diabetic Foot Ulcer Detection and Localization on Mobile Devices," *IEEE J. of Biomed. and Health Inform.*, vol. 23, no. 4, Jul. 2019.

[108] C.-T. Chen, *Linear system theory and design*, 3rd ed. Oxford University Press, 1998.

[109] M. Arjovsky, A. Shah, and Y. Bengio, "Unitary Evolution Recurrent Neural Networks," Nov. 2015.

[110] Q. V. Le, N. Jaitly, and G. E. Hinton, "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units," Apr. 2015.

[111] D. Haviv, A. Rivkind, and O. Barak, "Understanding and Controlling Memory in Recurrent Neural Networks," Feb. 2019.

[112] D. Sussillo and O. Barak, "Opening the Black Box: Low-Dimensional Dynamics in High-Dimensional Recurrent Neural Networks," *Neural Comp.*, vol. 25, no. 3, Mar. 2013.

[113] A. R. Voelker, "Dynamical Systems in Spiking Neuromorphic Hardware," Ph.D. dissertation, University of Waterloo, 2019.

[114] C. Hwang and M.-Y. Chen, "Analysis and parameter identification of time-delay systems via shifted Legendre polynomials," *Int. J. of Contr.*, vol. 41, no. 2, Feb. 1985.

[115] J. Jia and A. R. Benson, "Neural Jump Stochastic Differential Equations," May 2019.

[116] Y. Rubanova, R. T. Q. Chen, and D. Duvenaud, "Latent ODEs for Irregularly-Sampled Time Series," Jul. 2019.

[117] S. Chandar, C. Sankar, E. Vorontsov, S. E. Kahou, and Y. Bengio, "Towards Non-Saturating Recurrent Units for Modelling Long-Term Dependencies," *Proc. of the AAAI Conf. on AI*, vol. 33, Jul. 2019.

[118] L. Jing *et al.*, "Gated Orthogonal Recurrent Units: On Learning to Forget," *Neural Comp.*, vol. 31, no. 4, Apr. 2019.

[119] K. E. Helfrich, D. Willmott, and Q. Ye, "Orthogonal Recurrent Neural Networks with Scaled Cayley Transform," in *Int. Conf. On Mach. Learn. Res.*, Jul. 2018.

[120] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio, "Light Gated Recurrent Units for Speech Recognition," *IEEE Trans. on Emerg. Topics in Comp. Intell.*, vol. 2, no. 2, Apr. 2018.

[121] N. Vecoven, D. Ernst, and G. Drion, "A bio-inspired bistable recurrent cell allows for long-lasting memory," Jun. 2020.

[122] J. van der Westhuizen and J. Lasenby, "The unreasonable effectiveness of the forget gate," Apr. 2018.

[123] L. Wang, *Model Predictive Control System Design and Implementation using MATLAB.* Springer, 2009.

[124] A. D. Back and A. C. Tsoi, "Nonlinear system identification using discrete Laguerre functions," *J. of Syst. Eng.*, vol. 6, no. 6, 1996.

[125] L. Wang, "Discrete model predictive controller design using Laguerre functions," *J. of Proc. Contr.*, vol. 14, no. 2, Mar. 2004.

[126] A. R. Voelker, I. Kajic, and C. Eliasmith, "Legendre Memory Units : Continuous-Time Representation in Recurrent Neural Networks," in *Adv. in Neural Info. Proc. Syst.*, 2019.

[127] O. Azencot, N. B. Erichson, V. Lin, and M. M. Mahoney, "Forecasting Sequential Data using Consistent Koopman Autoencoders," in *37th Int. Conf. on Mach. Learn.*, 2020.

[128] B. Lusch, J. N. Kutz, and S. L. Brunton, "Deep learning for universal linear embeddings of nonlinear dynamics," *Nature Comm.*, vol. 9, no. 1, Dec. 2018.

[129] D. Gedon, N. Wahlström, T. B. Schön, and L. Ljung, "Deep State Space Models for Nonlinear System Identification," Mar. 2020.