

# DALINE: A Data-driven Power Flow Linearization Toolbox for Power Systems Research and Education

(User Manual for DALINE 1.1.5)

Mengshuo Jia, Wen Yi Chan, and Gabriela Hug

*Power Systems Lab, ETH Zürich*

*June 27, 2024*

Power flow linearization is a fundamental technique in power system operations, crucial for both academic research and industry applications. Despite its importance, many advanced linearization methods, particularly those driven by data, remain largely inaccessible, neither open-sourced nor integrated into widely used software platforms. To address this gap, we introduce Daline, an open-source MATLAB toolbox specifically designed for the power systems community. Daline provides a comprehensive suite of 57 linearization methods, including 53 data-driven techniques and 4 physics-driven approaches. Daline's robust capabilities cover a spectrum of functionalities: (1) Data Generation, (2) Data Pollution, (3) Data Cleaning, (4) Data Normalization, (5) Method Selection, (6) Method Customization, (7) Model Linearization, (8) Model Evaluation, and (9) Result Visualization. These functionalities enable users to execute complex tasks with minimal coding effort. This manual serves as an in-depth guide, providing a gentle introduction for new users and a detailed reference for advanced users, explaining Daline's features, functions, parameters, syntax, and practical applications, and all the other related contents. Our vision is to empower users by saving their valuable time and fostering innovation in power systems analysis through accessible, state-of-the-art linearization tools.

## Reminder to Users

Copying the code examples from the manual into MATLAB may introduce unwanted spaces, leading to syntax errors. Please check for suspicious spaces in strings before running the code. The easiest way is to directly use the code examples in the “examples” folder of the DALINE package without copying and pasting.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	License and Terms of Use . . . . .	3
1.3	Citing DALINE . . . . .	4
<b>2</b>	<b>Getting Started</b>	<b>5</b>
2.1	System Requirements . . . . .	5
2.2	Installation . . . . .	5
2.3	Major Functionalities . . . . .	8
2.4	Customization Approaches . . . . .	12
2.5	Customizable Parameters . . . . .	13
2.6	Daline Examples . . . . .	15
<b>3</b>	<b>Data Generating and Processing</b>	<b>16</b>
3.1	Data Generating ( <code>daline.generate</code> ) . . . . .	16
3.1.1	Input . . . . .	17
3.1.2	Output . . . . .	20
3.1.3	Examples . . . . .	21
3.1.4	Remarks . . . . .	21
3.2	Data Processing . . . . .	22
3.2.1	Check Data Format (Automatic) . . . . .	22
3.2.2	Add Data Noise ( <code>daline.noise</code> ) . . . . .	23
3.2.3	Add Data Outliers ( <code>daline.outlier</code> ) . . . . .	24
3.2.4	Filter Data Noise ( <code>daline.denoise</code> ) . . . . .	25
3.2.5	Filter Data Outliers ( <code>daline.deoutlier</code> ) . . . . .	26
3.2.6	Normalize Data ( <code>daline.normalize</code> ) . . . . .	28

3.3	All-in-one Command for Data Generating/Processing ( <code>daline.data</code> ) . . . . .	29
<b>4</b>	<b>Model Fitting and Testing</b>	<b>32</b>
4.1	All-in-one Command for Model Fitting/Testing ( <code>daline.fit</code> ) . . . . .	34
4.2	Least Squares Family . . . . .	38
4.2.1	Ordinary Least Squares ( <code>LS</code> ) . . . . .	39
4.2.2	Ordinary Least Squares with Generalized Inverse ( <code>LS_PIN</code> ) . . . . .	39
4.2.3	Least Squares with Singular Value Decomposition ( <code>LS_SVD</code> ) . . . . .	40
4.2.4	Least Squares with Complete Orthogonal Decomposition ( <code>LS_COD</code> ) . . . . .	41
4.2.5	Least Squares with Principal Component Analysis ( <code>LS_PCA</code> ) . . . . .	43
4.2.6	Least Squares with Huber Loss Function: a Direct Solution ( <code>LS_HBLD</code> ) . . . . .	45
4.2.7	Least Squares with Huber Loss Function: an Equivalent Solution ( <code>LS_HBLE</code> ) . . . . .	47
4.2.8	Least Squares with Huber Weighting Function ( <code>LS_HBW</code> ) . . . . .	49
4.2.9	Generalized Least Squares ( <code>LS_GEN</code> ) . . . . .	51
4.2.10	Total Least Squares ( <code>LS_TOL</code> ) . . . . .	53
4.2.11	Least Squares with Clustering ( <code>LS_CLS</code> ) . . . . .	54
4.2.12	Least Squares with Lifting Dimension: Lifting the Whole $\mathbf{x}$ Jointly ( <code>LS_LIFX</code> ) . . . . .	56
4.2.13	Least Squares with Lifting Dimension: Lifting the Elements of $\mathbf{x}$ Individually ( <code>LS_LIFXi</code> ) . . . . .	59
4.2.14	Weighed Least Squares ( <code>LS_WEI</code> ) . . . . .	61
4.2.15	Recursive Least Squares ( <code>LS_REC</code> ) . . . . .	63
4.2.16	Repeated Least Squares ( <code>LS_REP</code> ) . . . . .	65
4.3	Partial Least Squares Regression Family . . . . .	67
4.3.1	Ordinary Partial Least Squares with SIMPLS ( <code>PLS_SIM</code> ) . . . . .	67
4.3.2	Ordinary Partial Least Squares with SIMPLS Using Rank of $\mathbf{X}$ ( <code>PLS_SIMRX</code> ) . . . . .	68
4.3.3	Ordinary Partial Least Squares with NIPALS ( <code>PLS_NIP</code> ) . . . . .	69
4.3.4	Partial Least Squares Bundling Known/Unknown Variables and Replacing Slack Bus's Power Injection ( <code>PLS_BDL</code> ) . . . . .	70
4.3.5	Partial Least Squares Bundling Known/Unknown Variables ( <code>PLS_SIMY2</code> ) . . . . .	72
4.3.6	Partial Least Squares Bundling Known/Unknown Variables: the Open-source Ver- sion ( <code>PLS_BDLopen</code> ) . . . . .	72
4.3.7	Recursive Partial Least Squares with NIPALS ( <code>PLS_REC</code> ) . . . . .	73
4.3.8	Recursive Partial Least Squares with NIPALS with Forgetting Factors ( <code>PLS_RECW</code> ) . . . . .	75
4.3.9	Repeated Partial Least Squares with NIPALS ( <code>PLS_REP</code> ) . . . . .	77
4.3.10	Partial Least Squares with Clustering ( <code>PLS_CLS</code> ) . . . . .	79

4.4	Ridge Regression Family . . . . .	81
4.4.1	Ordinary Ridge Regression (RR) . . . . .	81
4.4.2	Ordinary Ridge Regression with the Voltage-angle Coupling (RR_VCS) . . . . .	83
4.4.3	Ordinary Ridge Regression with K-plane Clustering (RR_KPC) . . . . .	84
4.4.4	Locally Weighted Ridge Regression (RR_WEI) . . . . .	86
4.5	Support Vector Regression Family . . . . .	88
4.5.1	Ordinary Support Vector Regression: a Direct Solution (SVR) . . . . .	89
4.5.2	Support Vector Regression with Polynomial Kernel (SVR_POL) . . . . .	91
4.5.3	Support Vector Regression with Ridge Regression (SVR_RR) . . . . .	93
4.5.4	Support Vector Regression with Chance-constrained Programming (SVR_CCP) . . . . .	94
4.6	Linearly Constrained Programming Family . . . . .	97
4.6.1	General Inputs . . . . .	97
4.6.2	General Tips . . . . .	97
4.6.3	Linearly Constrained Programming with Box Constraints (LCP_BOX) . . . . .	97
4.6.4	Linearly Constrained Programming without Box Constraints (LCP_BOXN) . . . . .	99
4.6.5	Linearly Constrained Programming with Jacobian Guidance Constraints (LCP_JGD) . . . . .	100
4.6.6	Linearly Constrained Programming without Jacobian Guidance Constraints (LCP_JGDN) . . . . .	102
4.6.7	Linearly Constrained Programming with Coupling Constraints (LCP_COU and LCP_COU2) . . . . .	103
4.6.8	Linearly Constrained Programming without Coupling Constraints (LCP_COUN and LCP_COUN2) . . . . .	104
4.7	Distributionally Robust Chance-constrained Programming Family . . . . .	106
4.7.1	Moment-based Distributionally Robust Chance-constrained Programming with $\mathbf{X}$ as Random Variable (DRC_XM) . . . . .	108
4.7.2	Moment-based Distributionally Robust Chance-constrained Programming with $\mathbf{X}$ and $\mathbf{Y}$ as Random Variables (DRC_XYM) . . . . .	110
4.7.3	Divergence-based Distributionally Robust Chance-Constrained Programming with $\mathbf{X}$ and $\mathbf{Y}$ as Random Variables (DRC_XYD) . . . . .	112
4.8	Physical-model-informed Family . . . . .	115
4.8.1	DCPF (DC) . . . . .	115
4.8.2	DCPF with Ordinary Least Squares (DC_LS) . . . . .	116
4.8.3	Decoupled Linearized Power Flow (DLPF) . . . . .	116
4.8.4	DLPF with a Data-driven Correction (DLPF_C) . . . . .	117
4.8.5	Power Transfer Distribution Factor (PTDF) . . . . .	118

4.8.6	First-order Taylor Approximation (TAY)	119
4.9	Direct Solution Family	121
4.9.1	Direct QR Decomposition (QR)	121
4.9.2	Direct Left Division (LD)	122
4.9.3	Direct Generalized Inverse (PIN)	122
4.9.4	Direct Singular Value Decomposition (SVD)	123
4.9.5	Direct Complete Orthogonal Decomposition (COD)	124
4.9.6	Direct Principal Component Analysis (PCA)	125
<b>5</b>	<b>Performance Evaluation and Visualization</b>	<b>127</b>
5.1	Accuracy	127
5.2	Visualization of Computational Efficiency	139
5.2.1	Computational time rankings of multiple methods ( <code>daline.time</code> )	139
5.2.2	Computational time evolution curves of multiple methods ( <code>daline.time</code> )	141
<b>6</b>	<b>All-in-one Command for Daline (<code>daline.all</code>)</b>	<b>143</b>
6.1	Inputs	143
6.2	Outputs	143
6.3	Examples	144

# Chapter 1

## Introduction

### 1.1 Background

Due to the nonconvex nature of the alternating current (AC) power flow model, linearized power flow models are the major ones integral to daily operations, market clearing, and grid planning of power systems. These models facilitate trillion-dollar markets and affect every consumer globally. Enhancements in linear power flow models, either theoretical or technological, could deliver substantial societal and financial benefits.

Power flow linearization has been a focus of intense research for decades, aiming to enhance accuracy and efficiency. Recently, the surge in data-centric methodologies across science, engineering, technology, and societal applications has reinvigorated interest in this field, leading to the emergence of data-driven power flow linearization (DPFL). This field is rapidly evolving, driven by the widespread use of phasor measurement units, advanced communications infrastructures, sophisticated analysis techniques, real-time computing capabilities, and a keen interest in data-centric methods, making DPFL highly relevant.

Unlike traditional physics-driven (or say, model-driven) power flow linearization, DPFL methods often do not require pre-existing physical models of the power grid, relying instead on system measurements to train linear models. The advantages of DPFL include higher approximation accuracy due to its assumption-free, customizable nature (often significantly outperforming traditional methods), applicability in scenarios where physical parameters are missing, implicit consideration of power losses, integration of real-time measurements, and the accommodation of realistic impacts such as control actions and human behaviors. Further details on DPFL are available in our previous works [1], [2], and [3], which cover DPFL overviews, DPFL theories, and DPFL simulations, respectively.

With numerous DPFL approaches available, selecting the most suitable one can be challenging. Often, new methods are introduced with only selective comparisons highlighting their advantages, without thorough evaluations against all existing methods. Moreover, over 95% of DPFL approaches are not open-source, making them less accessible for research and education. Conversely, physics-driven approaches are typically more user-friendly, being integrated into common software and open-source toolkits, which continue to be popular among researchers, engineers, and educators.

To facilitate the adoption of DPFL, and more importantly, to make DPFL methods easily-accessible fundamental tools for the community, it is crucial to provide a comprehensive toolkit that includes all existing DPFL methods as built-in options. Such a toolkit would not only allow users to leverage the latest advancements in DPFL and tailor the tools to their specific needs but also enable the development of new methods based on existing ones. Although ten open questions in DPFL were already listed in [3], there are undoubtedly more that deserve attention. A well-equipped DPFL toolkit could significantly enhance this research, improving both accuracy and computational efficiency, thereby benefiting user projects, improving the societal and financial benefits of power systems, and ultimately reducing consumer electricity costs.

This is the motivation behind the development of **DALINE**, a **d**ata-driven power flow **l**inearization toolbox [4]. DALINE is a package of Matlab M-files. Its features include data generating, data polluting, data cleaning, and data normalizing, as well as model training, model testing, accuracy ranking, computational efficiency ranking/evolving, and multidimensional result visualizing. Over 55 linearization methods, either from the existing works or developed by us, are built in DALINE. Overall, DALINE was created to help users handle complex simulation and comparison tasks using just a few simple commands — typically, learning just three DALINE commands is enough for most; if users simply want to obtain a highly accurate linear power flow model for a given or standard power system, learning only one DALINE command is enough. The official website of this toolbox can be found at:

<https://www.shuo.science/daline>

DALINE was initially developed by Mengshuo Jia, Wen Yi Chan, and Gabriela Hug from the Power Systems Lab at ETH Zürich. The development of DALINE was supported by the Swiss National Science Foundation (No. 221126) and the Swiss National Centre of Competence in Research “Dependable, Ubiquitous Automation.”



## 1.2 License and Terms of Use

Starting from version 1.1.5, DALINE is distributed under the 3-Clause BSD License [5].

```
Copyright (c) 2024, Primary Developers to Daline (see authors.txt file).
All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

```
1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.
```

```
2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
```

```
3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.
```

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Note that the data generation functionality of DALINE is built on [MATPOWER](#) [6], and the built-in optimization methods of DALINE utilize toolboxes [CVX](#) [7] and [YALMIP](#) [8]<sup>1</sup>. To minimize installation and setup costs, DALINE includes redistributed versions of these toolboxes, in strict compliance with their redistribution licenses. License files for these toolboxes are also included in DALINE. Therefore, if users redistribute DALINE, they must adhere not only to DALINE's license but also to the licenses of these included toolboxes.

---

<sup>1</sup>Users will not interact with the underlying toolboxes, as they have been fully encapsulated within DALINE. Interaction with the interface of DALINE alone is sufficient for users to access all functionalities.

## 1.3 Citing Daline

While not mandated by the terms of the license, we do request that any publications derived from the use of DALINE explicitly acknowledge this by citing reference [4].

Mengshuo Jia, Wen Yi Chan, and Gabriela Hug. “Daline: A Data-driven Power Flow Linearization Toolbox for Power Systems Research and Education”. In: Under Review (2024)

```
@ARTICLE{Daline,
  author={Jia, Mengshuo and Chan, Wen Yi and Hug, Gabriela},
  title={Daline: A Data-driven Power Flow Linearization Toolbox for Power Systems
    Research and Education},
  year={2024},
}
```

(The above reference will be made public very soon. The update of the reference information can be found from the “Downloads” page at <https://www.shuo.science/daline>)

Furthermore, it is true that we have independently implemented the DPFL methods in DALINE given that over 95% of the DPFL literature does not provide open-source code, except for one notable exception which is frequently referenced in later sections. However, we strongly encourage users to cite the original sources of these methods when adopting them through DALINE, to honor the contributions of the researchers who originally proposed these approaches. To be clear, we emphasize the following points:

- We extensively cite the original papers (if any) of the DPFL methods in several ways, including, but not limited to: (i) in the table summarizing all built-in methods, (ii) within specific subsections introducing each method, and (iii) in the script for the main function of each method.
- We have attempted to accurately reproduce the methods in code based on the descriptions in the original papers. However, we cannot guarantee that the methods implemented in DALINE perfectly reflect the intentions of the original authors, due to various factors such as the lack of open-source code (with one exception) and the incomplete detail in many papers (for papers with blurring details, we have even implemented multiple versions of their methods, as users will see in Table 2.1 in the next chapter). We highlight the original papers in both the manual and the code scripts primarily to acknowledge their theoretical contributions, and we encourage users to do the same. Nonetheless, we do not claim — and indeed it is not feasible — that the built-in methods in DALINE are exact replicas of those envisioned by the original method creators.
- Not all methods built into DALINE originate from the literature. We also include a number of DPFL methods we developed, which have not yet been reported in existing DPFL studies. For these methods, no references are provided.

# Chapter 2

## Getting Started

### 2.1 System Requirements

DALINE is compatible with 64-bit versions of Linux, Mac OSX, and Windows. Initially, it was developed using MATLAB 2020b on a device equipped with an M1 chip. Generally, we recommend using the most recent version of MATLAB available to you. Ideally, the MATLAB version should not be more than five years old.

### 2.2 Installation

Installing DALINE is straightforward. After downloading DALINE, users simply need to run the setup file within the DALINE directory, and that is it — “if you find the installation easy, you are doing it right.” But for clarity, the detailed steps are outlined below:

- (1) Follow the download instructions on the [DALINE Home Page](#) or the [Github page](#) (Github page will coming soon). You should end up with a file named `dalineXXX.zip`, where `XXX` depends on the version of DALINE.
- (2) Unzip the downloaded file. Move the resulting `dalineXXX` directory to the location of your choice. These files should not need to be modified, so it is recommended that they be kept separate from your own code.
- (3) Open MATLAB and set the “Current Folder” to the directory containing `dalineXXX`. In the MATLAB command window, enter `daline_setup` to execute the setup and test suite, including installing DALINE, adding it to the path of MATLAB, and verifying that DALINE is properly installed and functioning (`daline_setup` will activate the parallel pool of MATLAB). Specifically,
  - Enter `daline_setup` or `daline_setup('fast')` to perform a complete setup, followed by a quick initial test of DALINE’s core functionalities, including data generation, pollution, cleaning, and processing, as well as modeling training and testing. Note that the process will skip the 17 built-in optimization-based linearization methods but only test the rest 40 built-in linearization

methods, as the optimization-based methods are more time-consuming. If necessary, users can separately test these methods using `daline_test`, as explained in Section 2.3.

- Enter `daline_setup('full')` to conduct a complete setup followed by a comprehensive testing process, including all optimization-based linearization methods. Note that this option may take significantly longer than `daline_setup` or `daline_setup('fast')`.

Note that the output of `daline_setup('full')` should look similar to the following, with variations based on the availability of optional packages, solvers, etc.

```
% Either daline_setup, daline_setup('fast'), or daline_setup('full')
>> daline_setup('full')
Setting Up Daline ..... Done!
Testing Data Generation ..... Succeeded!
Testing Data Pollution ..... Succeeded!
Testing Data Cleaning ..... Succeeded!
Testing Data Normalization .... Succeeded!
Testing Available Solvers ..... Done!
    Solver fminunc ..... Available!
    Solver fmincon ..... Available!
    Solver quadprog ..... Available!
    Solver Gurobi ..... Available!
    Solver SeDuMi ..... Available!
    Solver SDPT3 ..... Available!
Testing Linearization Methods ..... Start!
    Method LS ..... Available!
    Method QR ..... Available!
    Method LD ..... Available!
    Method PIN ..... Available!
    Method SVD ..... Available!
    Method PCA ..... Available!
    Method LS_SVD ..... Available!
    Method LS_PIN ..... Available!
    Method LS_COD ..... Available!
    Method COD ..... Available!
    Method LS_PCA ..... Available!
    Method LS_HBW ..... Available!
    Method LS_HBLD ..... Available!
    Method LS_HBLE ..... Available!
    Method LS_GEN ..... Available!
    Method LS_LIFX ..... Available!
    Method LS_LIFXi ..... Available!
```

```

Method LS_TOL ..... Available!
Method LS_CLS ..... Available!
Method LS_WEI ..... Available!
Method LS_REC ..... Available!
Method LS_REP ..... Available!
Method PLS_SIM ..... Available!
Method PLS_SIMRX ..... Available!
Method PLS_NIP ..... Available!
Method PLS_BDL ..... Available!
Method PLS_BDLY2 ..... Available!
Method PLS_BDLopen ..... Available!
Method PLS_REC ..... Available!
Method PLS_RECW ..... Available!
Method PLS_REP ..... Available!
Method PLS_CLS ..... Available!
Method RR ..... Available!
Method RR_WEI ..... Available!
Method RR_KPC ..... Available!
Method RR_VCS ..... Available!
Method SVR ..... Available!
Method SVR_POL ..... Available!
Method SVR_CCP ..... Available!
Method SVR_RR ..... Available!
Method LCP_BOXN ..... Available!
Method LCP_BOX ..... Available!
Method LCP_JGDN ..... Available!
Method LCP_JGD ..... Available!
Method LCP_COUN ..... Available!
Method LCP_COUN2 ..... Available!
Method LCP_COU ..... Available!
Method LCP_COU2 ..... Available!
Method DRC_XM ..... Available!
Method DRC_XYM ..... Available!
Method DRC_XYD ..... Available!
Method DC ..... Available!
Method DC_LS ..... Available!
Method DLPF ..... Available!
Method DLPF_C ..... Available!
Method PTDF ..... Available!
Method TAY ..... Available!

```

```

-----
Daline Setting Up and Initial Test Completed!
Available Methods: 57 Methods
Unavailable Methods: 0 Methods
Available Solvers: fminunc, fmincon, quadprog, Gurobi, SeDuMi, SDPT3
Please consult the user's manual for more information.
-----

```

**Remark 1:** DALINE includes some executable files from the redistributed version of **CVX**. Users' operating systems may not recognize the developer or may flag these files as untrusted. Consequently, users must manually authorize these files and configure their operating systems to allow their execution before or during setting up DALINE.

**Remark 2:** To minimize the installation and setup cost of users, solvers **SDPT3** and **SeDuMi** are included in DALINE via the redistributed version of CVX; **fminunc**, **quadprog**, and **fmincon** are built in the **MATLAB Optimization Toolbox**.

**Remark 3:** We assume that users have installed the MATLAB Optimization Toolbox and the Statistics and Machine Learning Toolbox when they installed MATLAB. To verify if these toolboxes are installed, users can enter `license('test', 'Optimization_Toolbox')` and `license('test', 'Statistics_Toolbox')` commands in the MATLAB command window. If the returned value is 1, the toolbox is installed and licensed. If users have not installed these toolboxes, nor any other general optimization solvers (such as Gurobi), we recommend installing them to enjoy the full functionality of DALINE. Without these toolboxes, the partial least squares-based methods and optimization-based methods may be affected. To install them, users have the following options:

- Rerun the MathWorks Installer to add any additional products not currently included in your installation. You do not need to reinstall MATLAB if you select the same installation folder. For instructions on installing products using the MathWorks Installer, see [here](#).
- Use the Add-On Explorer in MATLAB, if your license permits usage of it. For instructions on installing products using the Add-On Explorer, see [here](#).

**Remark 4:** If users have not installed **Gurobi**, they may need to do so manually if it is specifically required. However, this installation is optional, as only two linearization methods in DALINE require mixed-integer solvers like Gurobi (see Table 2.2 for more information). For those who wish to install Gurobi, we provide a “Gurobi\_Setup\_Tutorial.pdf” in the “docs” folder of DALINE, detailing how to set it up for free after connecting to the university network (yes, setting up Gurobi is not that easy). Please note that Gurobi should be used with YALMIP (again, see Table 2.2 for more information), as the redistributed version of CVX included in DALINE does not currently support commercial solvers.

## 2.3 Major Functionalities

DALINE's main functionality is to solve power flow linearization problems using data-driven techniques.

Consequently, the toolbox provides an entire suite of functions that allow users to:

1. Generate (optimal) power flow data from predefined power systems if needed,
2. Process artificial or realistic power system data into suitable inputs for data-driven linearization,
3. Choose training methods and test the resulting linearization models,
4. View, save, and analyze linearization results.

Given the appropriate inputs, these components can be run as standalone modules. However, they can also be “strung together” to form an entire, self-contained, but highly customizable DALINE pipeline. Specifically, the user interface of DALINE consists of a number of function wrappers, each containing several to dozens of functions. The main function wrappers of DALINE include:

- `daline.all`: Manages a complete cycle from data generation to results visualization.
- `daline.data`: Handles data generation, pollution, cleaning, and normalization.
- `daline.generate`: Generates training and testing data sets.
- `daline.noise`: Adds noise to data to simulate real-world conditions.
- `daline.outlier`: Introduces outliers into the data.
- `daline.denoise`: Removes noise from the data.
- `daline.deoutlier`: Filters outliers from the data.
- `daline.normalize`: Normalizes data sets with the unit energy normalization.
- `daline.fit`: Trains and tests models based on the data and the selected method.
- `daline.rank`: Compares and ranks different training methods.
- `daline.time`: Assesses the computing efficiency of single or multiple training methods.
- `daline.plot`: Visualizes results in various dimensions with different themes.

A key feature of DALINE is its support for an extensive collection of linearization methods. The majority of these methods are data-driven (some also have a physics-informed foundation), while others are purely physics-based, serving as benchmarks. The built-in linearization methods are listed in Table 2.1.

Table 2.1: Built-in linearization approaches in DALINE

Data-driven power flow linearization approaches			
Ref.	Built-in method name	training algorithm	supporting technique
[9]	LS	Ordinary Least Squares	-
[10, 11]	LS_SVD	Least Squares with Singular Value Decomposition	-
[10, 11]	LS_COD	Least squares with Complete Orthogonal Decomposition	-
[12, 13]	LS_HBLD	Least Squares with Huber Loss - Solved Directly	-
[12, 13]	LS_HBLE	Least Squares with Huber Loss - Equivalent Convex Transformation	-
[14]	LS_TOL	Total Least Squares	-
[15]	LS_CLS	Clustering-based Least Squares	-
[16, 17]	LS_LIFX	Ordinary Least Squares	Dimension Lifting; Lift $\mathbf{x}$
[16]	LS_LIFXi	Ordinary Least Squares	Dimension Lifting; Lift the $i$ -th dimension of $\mathbf{x}$
[9]	LS_WEI	Ordinary Least Squares	Voltage Squaring; Forgetting Factor
[18]	LS_REC	Recursive Least Squares	-
-	LS_REP	Repeated Least Squares	-
-	LS_PIN	Least Squares with Pseudoinverse	-
-	LS_PCA	Least Squares with Principal Component Analysis	-
-	LS_GEN	Generalized Least Squares with Pseudoinverse	-
-	LS_HBW	Least Squares with Huber Weighting Function	-
[19]	PLS_SIM	Ordinary Partial Least Squares with SIMPLS	-
[19]	PLS_SIMRX	Ordinary Partial Least Squares with SIMPLS using rank of $\mathbf{X}$	-
[20]	PLS_BDOpen	Ordinary Partial Least Squares (Open-source Code of [20])	Bundle Variables; Replace Slack Bus's Injection
[20]	PLS_BDL	Ordinary Partial Least Squares	Bundle Variables; Replace Slack Bus's Injection
[20]	PLS_BDLY2	Ordinary Partial Least Squares	Bundle Variables
[21]	PLS_REC	Recursive Partial Least Squares	-
[21]	PLS_RECW	Recursive Partial Least Squares	Forgetting Factor
[21]	PLS_NIP	Ordinary Partial Least Squares with NIPALS	-
-	PLS_CLS	Clustering-based Partial Least Squares	-
-	PLS_REP	Repeated Partial Least Squares with NIPALS	-
[22]	RR	Ordinary Ridge Regression	-
[22]	RR_VCS	Ordinary Ridge Regression	Voltage-angle Coupling; Voltage Squaring
[23]	RR_KPC	Clustering-based Ridge Regression	Voltage Squaring
[24]	RR_WEI	Locally Weighted Ridge Regression	-
[25]	SVR	Ordinary Support Vector Regression	Voltage Squaring
[26]	SVR_CCP	Chance-constrained Programming	Grid Topology Integration
[27, 28]	SVR_POL	Support Vector Regression with Kernels	-
[29]	SVR_RR	Support Vector Regression with Regularization	-
[30]	LCP_BOX	Linearly Constrained Program with Bound Constraints	Voltage Squaring
[30]	LCP_COU or LCP_COU2	Linearly Constrained Program with Coupling Constraints	Grid Topology Integration; Voltage Squaring
[14]	LCP_JGD	Linearly Constrained Program with Structure Constraints	Bundle Known and Unknown Variables
[31]	DRC_XM	Moment-based DRCC <sup>§</sup> Program: Random $\mathbf{X}$	-
[31]	DRC_XYM	Moment-based DRCC <sup>§</sup> Program: Random $\mathbf{X}$ and $\mathbf{Y}$	-
[31]	DRC_XYD	Divergence-based DRCC <sup>§</sup> Program: Random $\mathbf{X}$ and $\mathbf{Y}$	-
[32, 33]	DC_LS	Ordinary Least Squares	Physical Model's Coefficient Optimization via LS
[34]	DLPF_C	Ordinary Least Squares	Physical Model's Error Correction via QR
[30]	LCP_BOXN	Linearly Constrained Program without Bound Constraints	Voltage Squaring
[30]	LCP_COUN or LCP_COU2N	Linearly Constrained Program without Coupling Constraints	Voltage Squaring
[14]	LCP_JGDN	Linearly Constrained Program without Structure Constraints	Bundle Known and Unknown Variables
-	QR	Direct QR Decomposition	-
-	LD	Direct Left Division	-
-	PIN	Direct Generalized Inverse	-
-	SVD	Direct Singular Value Decomposition	-
-	COD	Direct Complete Orthogonal Decomposition	-
-	PCA	Direct Principal Component Analysis	-
Physics-driven power flow linearization approaches			
Ref.	Abbreviation	Approach	
-	DC	Classic Direct Current Model	
-	PTDF	Classic Power Transfer Distribution Factors	
-	TAY	Warm-start 1st order Taylor approximation	
[35]	DLPF	Decoupled Linearized Power Flow	

§ : “DRCC” refers to “Distributionally Robust Chance-constrained”



Table 2.2: Summary of Methods that Require Optimization Toolkits and Solvers (SDPT3 and SeDuMi are included in DALINE via CVX; fminunc, quadprog and fmincon are built in the MATLAB Optimization Toolbox; for Gurobi, users need to install it manually if needed)

Method	Required Toolkits	Applicable Solvers	Recommendation(s)
LS_HBLD	MATLAB Optimization Toolbox	fminunc	-
LS_HBLE	YALMIP or CVX	fmincon, Gurobi	YALMIP + fmincon
LS_WEI	YALMIP or CVX	quadprog, Gurobi, SDPT3, SeDuMi	CVX + SeDuMi
SVR	YALMIP or CVX	quadprog, Gurobi	YALMIP + quadprog
SVR_RR	YALMIP or CVX	quadprog, Gurobi	YALMIP + quadprog
SVR_CCP	YALMIP or CVX	Gurobi	YALMIP + Gurobi
DRC_XYD	YALMIP or CVX	Gurobi	YALMIP + Gurobi
DRC_XM	CVX	SDPT3, SeDuMi	CVX + SeDuMi
DRC_XYM	CVX	SDPT3, SeDuMi	CVX + SeDuMi
LCP_BOX	CVX	quadprog, SDPT3, SeDuMi	SeDuMi
LCP_BOXN	CVX	quadprog, SDPT3, SeDuMi	SeDuMi
LCP_JGD	CVX	SDPT3	SDPT3
LCP_JGDN	CVX	SDPT3	SDPT3
LCP_COU	CVX	quadprog, SDPT3, SeDuMi	quadprog
LCP_COUN	CVX	quadprog, SDPT3, SeDuMi	quadprog
LCP_COU2	CVX	quadprog, SDPT3, SeDuMi	quadprog
LCP_COUN2	CVX	quadprog, SDPT3, SeDuMi	quadprog

**Remark:** The solvers listed in Table 2.2 are not mandatory; they are integrated into DALINE to simplify the installation and setup process for users. Users are free to use any solver they are familiar with, provided it is capable of addressing the corresponding problems and is already installed and functioning. For example, SeDuMi can be replaced with other cone optimization solvers, and Gurobi can be replaced with CPLEX, etc.

Additionally, as mentioned previously, users can quickly test specific methods of interest using `daline_test` to verify their availability after DALINE has been automatically set up; see the examples below. Normally, all non-optimization-based methods should be ready to use immediately after setup. However, the availability of optimization-based methods depends on the solvers (and optimization toolboxes) installed. By integrating the re-distribution versions of toolboxes and solvers, DALINE already reduced the need for additional installations of external resources. Typically, only two optimization-based methods that utilize integer variables, `SVR_CCP` and `DRC_XYD`, may be unavailable if solvers such as Gurobi are not installed manually.

```
% Test all the built-in methods (57 methods in total)
>> daline_test % Or daline_test('full')
```

```
% Test only the non-optimization-based methods (40 methods in total)
>> daline_test('fast')
```

```
% Test methods of interest, such as 'PLS_REC' and 'RR_KPC'
>> daline_test({'PLS_REC'; 'RR_KPC'})
```

## 2.4 Customization Approaches

While DALINE supports a wide array of function wrappers, this toolbox is meticulously designed to ensure consistency across its entire range of function wrappers, thereby enhancing user experience. Central to DALINE's design philosophy are two universal and intuitive methods for customization: **name-value pairs** and the **option structure**. This section aims to provide users with an overview of these customization techniques, which are applicable throughout the toolbox. In-depth explorations of individual wrappers and their unique parameters will be covered in the sections that follow.

### Name-value pairs

Function parameters can be specified as name-value pairs, providing a straightforward way to customize functionality. In total, there are more than 300 name-value pairs embedded in DALINE. However,

1. **Order Independence:** The name-value pairs can be provided in any sequence, allowing users to organize their code in the most logical and intuitive manner. This flexibility ensures that the order of parameters does not impact the function's execution.
2. **Selective Specification:** It is not necessary to specify all possible parameters. Users only need to provide those parameters they wish to customize. The toolbox automatically uses default settings for any parameters not explicitly mentioned, simplifying initial use and reducing the need for extensive configuration.

The following example highlights just three of the 49 adjustable name-value pairs available within `daline.data`:

```
>> data = daline.data('case.name', 'case118', 'data.program', 'acpf', 'data.baseType', '
    TimeSeriesRand');
```

- 'case.name': name of a parameter, specifies the power system case to study.
- 'case118': value for 'case.name' parameter, i.e., studying the IEEE 118-bus system.
- 'data.program': name of another parameter, specifying how power flow will be computed.
- 'acpf': value for 'data.program', i.e., using the AC power flow calculation.
- 'data.baseType': name of another parameter, specifying how the data will be generated.
- 'TimeSeriesRand': value for 'data.baseType', i.e., generated as a random time series.

Indeed, managing multiple parameters, particularly when comparing different linearization methods with various hyperparameters, can be tedious when using name-value pairs. To streamline this process, DALINE offers a default parameter sheet called `func_default_option_category`. This sheet organizes all adjustable parameters and options, complete with their default values and descriptions. Users can effortlessly configure multiple parameters simultaneously by adjusting the default values in this sheet. Setting parameters in this sheet can also help avoid conflicts that occur when different approaches adopt the same parameter names.

### Option structure

For a more streamlined way to specify parameters, DALINE allows using an option structure. This structure can be pre-configured with desired parameters and their values, and then passed to functions as a single argument. This method shares the same benefits of flexibility and selective specification as

name-value pairs. Yet, by defining an option structure with custom settings, users encapsulate all their configurations in one place. This structure can then be reused across multiple function calls, promoting code reusability and reducing redundancy. E.g.,

```
>> opt = daline.setopt('case.name', 'case118', 'data.program', 'acpf', 'data.baseType',
    'TimeSeriesRand');
>> data = daline.data(opt);
```

For some wrapper functions, they require a primary argument (and/or a secondary argument) followed by name-value pairs. E.g., the first argument of `daline.all` should refer to a power system case (either internal cases of MATPOWER or external cases defined by users):

```
>> model = daline.all('case118', 'data.baseType', 'TimeSeriesRand', 'method.name', 'RR'
    );
% Regardless of the linearization method used, model.Beta is always the outputted
    linear model => a mapping matrix that projects the (selected) independent
    variables to the (selected) dependent variables. This linear model can be
    readily used for many applications.
```

For such wrappers, users can still use `daline.setopt` to simplify the arguments of these functions, e.g.,

```
>> opt = daline.setopt('data.baseType', 'TimeSeriesRand', 'method.name', 'RR');
>> model = daline.all('case118', opt);
```

**Remark:** When users need to adjust numerous parameters, manually typing the name-value pairs of these parameters and entering them into `daline.rank` can be cumbersome and inelegant. In such cases, users are suggested to directly modify the file that holds the default parameters, named `func_default_option_category`. After setting these default parameters to meet their requirements, users can then straightforwardly call the wrappers without the need to manually input a large number of name-value pairs themselves.

## 2.5 Customizable Parameters

Given that DALINE supports hundreds of parameters, understanding the available parameters for customization is crucial. The `daline.getopt` function provides information about the adjustable options and default settings, aiding users in fine-tuning their analyses. The usage of `daline.getopt` is straightforward. For instance, if users wish to explore the parameters associated with data generation and the addition of outliers, the following code will retrieve these parameters within an options structure:

```
>> opt = daline.getopt('generate data', 'add outlier');
```

It is important for users to be familiarized with the names of the parameter categories before attempting to access them via `daline.getopt`, e.g., ‘generate data’ and ‘add outlier’ shown above. Each category name can be utilized as an argument in `daline.getopt`, which is capable of accepting multiple categories

simultaneously. Table 2.3 provides a summary of all available parameter categories. Furthermore, the nomenclature of every built-in method listed in Table 2.1 is also recognized as a valid category name when interacting with `daline.getopt`, allowing users to retrieve the hyperparameters of the specified method. E.g.,

```
>> opt = daline.getopt('RR_KPC');
```

Table 2.3: Parameter Categories in DALINE

Category	Meaning
'system'	Default options related to system case .
'generate data'	Default options related to data generation. Settings include training and testing data points, redundancy, power flow program, data type, ranges, switches for parallel computing, and other various data settings.
'add outlier'	Default options related to adding outlier. Settings include switches for random fixedness, training/testing data switches, outlier percentages, and others.
'add noise'	Default options related to adding noise. Settings include switches for random fixedness, training/testing noise addition, signal-to-noise ratio, etc.
'filter outlier'	Default options related to filtering outlier. Settings include switch for training/testing data, filter method, tolerance level, etc.
'filter noise'	Default options related to filtering noise. Settings include switches for training/testing data, noise degree estimation, dynamic estimation model, initialization settings, etc.
'normalize data'	Data normalization related default options.
'mpc index'	Index definition for built-in system cases of MATPOWER.
'method'	Default linearization method selection. Settings include the default method name and the entire list of supported methods.
'warning'	Default switches for turning warning on/off for all methods.
'variable'	Global variable settings for all methods, including predictor, response, various flags for lifting dimensions, etc.
Any Built-in Method Name listed in Table 2.1	Default hyperparameters of the specified method.

It is crucial to understand that although `daline.getopt` is capable of accepting multiple categories in a single call, it is restricted to handling just one method name at a time. This limitation is in place because various methods might share parameters, leading to potential ambiguities. To prevent any confusion, `daline.getopt` is designed to process only a single method name per invocation. However, this constraint does not apply to categories; you can input as many categories as needed without restriction. E.g.,

```
% Correct:
>> opt = daline.getopt('generate data', 'RR_KPC');

% Incorrect:
>> opt = daline.getopt('generate data', 'RR_KPC', 'RR');

% Incorrect:
>> opt = daline.getopt('RR_KPC', 'RR');
```

## 2.6 Daline Examples

As users have observed, several examples of DALINE are already provided. Many more additional examples will be introduced in the subsequent chapters. The scripts for all these examples are stored in the “examples” folder within the DALINE package.

## Chapter 3

# Data Generating and Processing

Power system measurements are essential for implementing data-driven linearization methods. **For users without historical data** for a targeted power system, DALINE provides the functionality to create synthetic data specifically designed for the given system. This system may be defined using either a standard MATPOWER case or a user-supplied custom case that complies with the MATPOWER format. DALINE further enhances the realism of this synthetic data by introducing noise and/or outliers if needed. **For those who already possess data**, DALINE offers tools to refine it, including filtering out noise and outliers. It also provides data normalization capabilities to facilitate more effective model fitting. The above operations are executed by DALINE's data generation and processing modules, which are elaborated upon in this chapter. Specifically, this chapter begins by introducing the individual function wrappers associated with data generation and processing, including

- `daline.generate`: Generates training and testing data sets.
- `daline.noise`: Adds noise to data to simulate real-world conditions.
- `daline.outlier`: Introduces outliers into the data.
- `daline.denoise`: Removes noise from the data.
- `daline.deoutlier`: Filters outliers from the data.
- `daline.normalize`: Normalizes data sets with the unit energy normalization.

To further enhance user convenience, DALINE also offers:

- `daline.data`: Handles data generation, pollution, cleaning, and normalization.

`daline.data` integrates all functions associated with data generation and processing, which will be discussed in detail at the end of this chapter.

**It is important to note that the outputs of all the aforementioned wrappers are consistent: a structured data format. This format will be elaborated upon in Section 3.2.1.**

### 3.1 Data Generating (`daline.generate`)

In DALINE, the task of data generation is accomplished through the `daline.generate` wrapper. The fundamental operations of `daline.generate` are outlined as follows:

1. **Load the Case:** `daline.generate` initiates the process by loading the power system case specified by the user. This can be either a standard case included with MATPOWER or a custom case provided by the user that adheres to the MATPOWER format.
2. **Alter Operating Points:** Subsequently, `daline.generate` modifies various parameters such as load conditions, power generation, and nodal voltage magnitudes of the system, in accordance with user-defined settings and strategies. This step is aimed at generating a diverse set of potential operating points for the system.
3. **Compute Power Flows:** For each generated operating point, `daline.generate` computes the power flow results utilizing one of the power flow solvers specified by the user. This produces a comprehensive dataset representing power flow measurements under varied conditions.
4. **Organize Data:** In the final step, `daline.generate` divides the power flow data into training and testing sets as per the user's configuration. These datasets are then organized into a structured format, grouping different types of data (such as voltage magnitudes, voltage angles, active branch flows, etc.) for ease of use.

### 3.1.1 Input

The `daline.generate` function accepts input either as name-value pairs or as an option structure that contains the user's name-value configurations. The default name-value parameters for `daline.generate` are detailed in Table 3.1, which can all be treated as inputs.

Table 3.1: Default name-value input arguments for `daline.generate`.

Parameter	Format	Default	Description
<code>case.name</code>	character	'case39'	Name of a MATPOWER case of a power system. See Appendix D.3 in [36] for the full list of MATPOWER cases. This parameter is exclusive and cannot be used concurrently with the parameter <code>case.mpc</code> ; only one of the two can be set at a time.
<code>case.mpc</code>	struct	<code>mpc</code> of case39 in MATPOWER	A power system defined in the standard <code>mpc</code> structure. See Section 3.1 in [36] for further details on the structure of <code>mpc</code> case. This parameter is exclusive and cannot be used concurrently with the parameter <code>case.name</code> ; only one of the two can be set at a time.
<code>num.trainSample</code>	integer	300	Number of training data points.
<code>num.testSample</code>	integer	200	Number of testing data points.

Table 3.1 continued from previous page

Parameter	Format	Default	Description
<code>num.redundant</code>	integer	50	Redundant samples to account for potential failure if computing (optimal) power flows (though this is not common, it may happen if users, e.g., increase the loading condition too much, do not allow the change in voltage magnitudes in certain cases, etc.). Unless the number of failures is greater than <code>num.redundant</code> , users always get the expected numbers of training and testing data points.
<code>data.parallel</code>	binary	1	Use MATLAB Parallel Computing Toolbox for speedup: 1 for yes, 0 for no.
<code>data.program</code>	character	'acpf'	Data generation method: 'acopf', 'acpf', or 'dcopf_acpf'. Note that 'dcopf_acpf' first runs DCOPF for generation dispatch then runs ACPF to mock the real AC power flows after accepting the DCOPF dispatch plan.
<code>data.baseType</code>	character	'TimeSeriesRand'	Type of data generation: 'Random', 'TimeSeries', 'TimeSeriesRand'. In the type 'Random', the random factor in changing operating points (a.k.a., changing factor) follows a standard uniform distribution. In the type 'TimeSeries', the changing factor follows a given time series curve. In the type 'TimeSeriesRand', the changing factor follows a given time series curve plus randomness from a standard uniform distribution.
<code>data.curvePlot</code>	binary	0	Plot the time series curve for 'TimeSeries' or 'TimeSeriesRand': 1 for yes, 0 for no.
<code>data.fixRand</code>	binary	0	Fix randomness during data generation: 1 for yes, 0 for no.
<code>data.fixSeed</code>	integer	88	Seed to fix randomness.
<code>load.distribution</code>	character	'uniform'	The distribution of the randomness for the loading; it can be 'uniform' or 'normal'.
<code>load.amplifyFactor</code>	float	0.9	Factor to multiply each nodal load to regulate load level globally.
<code>load.smallValue</code>	float	0.05	Value added to each nodal load to avoid all-zero power injections.



Table 3.1 continued from previous page

Parameter	Format	Default	Description
<code>load.upperRange</code>	float	1.2	Upper range for nodal load change in 'Random' mode. When the normal distribution is chosen, the randomness can only be tuned but not strictly limited by the bound owing to the nature of the distribution.
<code>load.lowerRange</code>	float	0.8	Lower range for nodal load change in 'Random' mode. When the normal distribution is chosen, the randomness can only be tuned but not strictly limited by the bound owing to the nature of the distribution.
<code>load.upperRangeTime</code>	float	1.05	Upper range for nodal load change in 'TimeSeries-Rand' mode. When the normal distribution is chosen, the randomness can only be tuned but not strictly limited by the bound owing to the nature of the distribution.
<code>load.lowerRangeTime</code>	float	0.95	Lower range for nodal load change in 'TimeSeries-Rand' mode. When the normal distribution is chosen, the randomness can only be tuned but not strictly limited by the bound owing to the nature of the distribution.
<code>load.baseLoadCurve</code>	array	[1.0300,...,1.0500]	Default daily base load curve with 48 points for 'TimeSeries' or 'TimeSeriesRand', i.e., the time series curve aforementioned. This curve was modified from a national half-hourly load curve of France on Aug.16, 2023.
<code>load.timeStart</code>	character	'19:00'	Start for the time window of interest when using 'TimeSeries' or 'TimeSeriesRand' to generate data. Half-hour resolution, i.e., only formats like '5:00' and '23:30' are accepted.
<code>load.timeEnd</code>	character	'22:00'	End for the time window of interest when using 'TimeSeries' or 'TimeSeriesRand' to generate data. Half-hour resolution, i.e., only formats like '5:00' and '23:30' are accepted.
<code>voltage.varyIndicator</code>	binary	0	Randomly change voltage magnitudes of PV buses: 1 for yes, 0 for no.
<code>voltage.distribution</code>	character	'uniform'	The distributon of the randomness for the voltage magnitudes; it can be 'uniform' or 'normal'.

Table 3.1 continued from previous page

Parameter	Format	Default	Description
<code>voltage.upperRange</code>	float	0.05	Upper range for random voltage magnitude change at PV buses. When the normal distribution is chosen, the randomness can only be tuned but not strictly limited by the bound owing to the nature of the distribution.
<code>voltage.lowerRange</code>	float	-0.05	Lower range for random voltage magnitude change at PV buses. When the normal distribution is chosen, the randomness can only be tuned but not strictly limited by the bound owing to the nature of the distribution.
<code>voltage.refAngle</code>	float	10	Small value added to the reference angle of the slack bus to avoid all-zero angles.
<code>gen.varyIndicator</code>	binary	1	Changes power generations simultaneously with the change in load in 'acpf': 1 for yes, 0 for no. For 'acopf' or 'dcopf_acpf', power generations will change automatically to meet the demand (and power loss).
<code>gen.smallValue</code>	float	0.08	Value added to each generator output in 'acpf' when changing power generations, in order to avoid all-zero power generations.

### 3.1.2 Output

As shown in the examples previously, the output from `daline.generate` is `data`, which is organized into a structured format to facilitate ease of access. The dataset is segmented into three primary components: `train`, `test`, and `mpc`.

- **Training Dataset (`train`):** This field contains a diverse array of data points tailored for the training process of linear power flow models. It covers various conditions reflective of real-world power system scenarios, based on users' settings when calling `daline.generate`.
- **Testing Dataset (`test`):** Aimed at linear model testing, this field includes data points essential for assessing model performance. The data points here are structured in the same way as the data points in the `train` field.
- **MATPOWER Case Data (`mpc`):** Comprising detailed power system configuration information, including buses, lines, generators, and loads.

Below is a hierarchical overview of the structure of `data`, including primary fields and their respective sub-fields ( $N_{train}$ : number of training data points, i.e., `num.trainSample`;  $N_{test}$ : number of testing data points, i.e., `num.testSample`;  $N_{bus}$ : number of buses in the power system;  $N_{line}$ : number of lines in the system). The nodal data are arranged in ascending order based on the bus indices from 1 to  $N_{bus}$ , and the branch data are organized according to the line indices from 1 to  $N_{line}$ .

- `data.train`
  - `data.train.P` ( $N_{train} \times N_{bus}$  matrix): nodal active power injections;
  - `data.train.Q` ( $N_{train} \times N_{bus}$  matrix): nodal reactive power injections;
  - `data.train.Vm` ( $N_{train} \times N_{bus}$  matrix): nodal voltage magnitudes;
  - `data.train.Vm2` ( $N_{train} \times N_{bus}$  matrix): square of nodal voltage magnitudes;
  - `data.train.Va` ( $N_{train} \times N_{bus}$  matrix): nodal voltage angle;
  - `data.train.PF` ( $N_{train} \times N_{line}$  matrix): active branch flows from the “From-bus”;
  - `data.train.PT` ( $N_{train} \times N_{line}$  matrix): active branch flows from the “To-bus”;
  - `data.train.QF` ( $N_{train} \times N_{line}$  matrix): reactive branch flows from the “From-bus”;
  - `data.train.QT` ( $N_{train} \times N_{line}$  matrix): reactive branch flows from the “To-bus”;
- `data.test`
  - `data.test.P` ( $N_{test} \times N_{bus}$  matrix): nodal active power injections;
  - `data.test.Q` ( $N_{test} \times N_{bus}$  matrix): nodal reactive power injections;
  - `data.test.Vm` ( $N_{test} \times N_{bus}$  matrix): nodal voltage magnitudes;
  - `data.test.Vm2` ( $N_{test} \times N_{bus}$  matrix): square of nodal voltage magnitudes;
  - `data.test.Va` ( $N_{test} \times N_{bus}$  matrix): nodal voltage angle;
  - `data.test.PF` ( $N_{test} \times N_{line}$  matrix): active branch flows from the “From-bus”;
  - `data.test.PT` ( $N_{test} \times N_{line}$  matrix): active branch flows from the “To-bus”;
  - `data.test.QF` ( $N_{test} \times N_{line}$  matrix): reactive branch flows from the “From-bus”;
  - `data.test.QT` ( $N_{test} \times N_{line}$  matrix): reactive branch flows from the “To-bus”;
- `data.mpc` (the standard MATPOWER case data)

### 3.1.3 Examples

```
>> data = daline.generate('case.name', 'case118', 'data.program', 'acpf', 'data.
    baseType', 'TimeSeriesRand');
```

```
>> opt = daline.setopt('case.name', 'case39', 'num.trainSample', 500, 'num.testSample',
    300);
>> data = daline.generate(opt);
```

```
% Assume the user already loaded a pre-defined mpc
>> opt = daline.setopt('case.mpc', mpc, 'load.timeStart', '7:00', 'load.timeEnd', '9:30
    ');
>> data = daline.generate(opt);
```

### 3.1.4 Remarks

- When enabling parallel computation for the first time, MATLAB may require additional time to initialize the parallel computing toolkit (if already installed). Yet, for the subsequent data generation, parallel computation can significantly speed up the data generation process. E.g., it only takes several seconds to generate 500 scenarios for a 1354-bus system on a personal laptop with a M1 chip (8-core) and 16 GB RAM.

- Some data-driven linearization methods cannot handle all-zero columns in the training dataset. Using `load.smallValue`, `voltage.refAngle`, and `gen.smallValue` can avoid such situations.
- Solving ACOPF with fixed PV voltage magnitudes presents significant challenges and is prone to a high incidence of failures.

## 3.2 Data Processing

DALINE can process both synthetically generated datasets by DALINE itself and external datasets provided by users. It offers the ability to introduce or filter data noise and outliers, and normalize data, which are crucial for either testing the robustness of models, cleaning the data beforehand, or standardizing the range of data features.

### 3.2.1 Check Data Format (Automatic)

The required input for all data processing function wrappers within DALINE is a structure, referred to as `data`, which must adhere to DALINE's standardized data format. Should the `data` structure be produced by DALINE itself, it inherently meets the necessary format specifications. However, if the `data` structure is provided directly by users, its format will be automatically verified upon using any of DALINE's data processing (as well as model training/testing) function wrappers. Specifically, the input `data` must be a structure (`struct` in MATLAB) with the following primary fields:

- `data.train`
- `data.test`
- `data.mpc` (This should be a standard MATPOWER case [36])

Structure `data.train` should contain the following sub-fields:

- **Mandatory Sub-fields:**
  - `data.train.P` ( $N_{train} \times N_{bus}$  matrix): nodal active power injections;
  - `data.train.Q` ( $N_{train} \times N_{bus}$  matrix): nodal reactive power injections;
- **Optional Sub-fields:** At least one of the following:
  - `data.train.Vm` ( $N_{train} \times N_{bus}$  matrix): nodal voltage magnitudes;
  - `data.train.Vm2` ( $N_{train} \times N_{bus}$  matrix): square of nodal voltage magnitudes;
  - `data.train.Va` ( $N_{train} \times N_{bus}$  matrix): nodal voltage angle;
  - `data.train.PF` ( $N_{train} \times N_{line}$  matrix): active branch flows from the "From-bus";
  - `data.train.PT` ( $N_{train} \times N_{line}$  matrix): active branch flows from the "To-bus";
  - `data.train.QF` ( $N_{train} \times N_{line}$  matrix): reactive branch flows from the "From-bus";
  - `data.train.QT` ( $N_{train} \times N_{line}$  matrix): reactive branch flows from the "To-bus";

Structure `data.test` should contain the following sub-fields:

- **Mandatory Sub-fields:**
  - `data.test.P` ( $N_{test} \times N_{bus}$  matrix): nodal active power injections;
  - `data.test.Q` ( $N_{test} \times N_{bus}$  matrix): nodal reactive power injections;
- **Optional Sub-fields:** At least one of the following:
  - `data.test.Vm` ( $N_{test} \times N_{bus}$  matrix): nodal voltage magnitudes;

- `data.test.Vm2` ( $N_{test} \times N_{bus}$  matrix): square of nodal voltage magnitudes;
- `data.test.Va` ( $N_{test} \times N_{bus}$  matrix): nodal voltage angle;
- `data.test.PF` ( $N_{test} \times N_{line}$  matrix): active branch flows from the “From-bus”;
- `data.test.PT` ( $N_{test} \times N_{line}$  matrix): active branch flows from the “To-bus”;
- `data.test.QF` ( $N_{test} \times N_{line}$  matrix): reactive branch flows from the “From-bus”;
- `data.test.QT` ( $N_{test} \times N_{line}$  matrix): reactive branch flows from the “To-bus”;

For interested readers only: the built-in function `func_check_dataFormat` ensures that the data provided to the DALINE toolbox adheres to the required standard format; an output of `isValid` equal to 1 means that `data` has met the format validation criteria.

```
>> isValid = func_check_dataFormat(data);
```

### 3.2.2 Add Data Noise (`daline.noise`)

DALINE can introduce white Gaussian noise, a general form of noise observed in electrical measurements. This is facilitated through the `daline.noise` wrapper. The basic idea is directly adding small values (following a Gaussian distribution with a zero mean) to the training and/or testing datasets, based on the user’s configuration. The primary and mandatory argument of `daline.noise` is `data` that satisfies the data format of DALINE, as explained above. Usage examples are provided below, and the default parameters for `daline.noise`, all of which are modifiable, are detailed in Table 3.2.

Table 3.2: Default Name-value Input Arguments for `daline.noise`.

Parameter	Format	Default	Description
<code>data.fixRand</code>	binary	0	Fixes randomness when introducing noise: 1 for yes, 0 for no.: 1 for yes, 0 for no.
<code>data.fixSeed</code>	integer	88	The seed used to fix the randomness in adding noise.
<code>noise.switchTrain</code>	binary	1	Activates adding noise to the training data: 0 for no, 1 for yes (1 by default when using <code>daline.noise</code> , but 0 by default when using <code>daline.data</code> ).
<code>noise.switchTest</code>	binary	0	Activates adding noise to the testing data: 0 for no, 1 for yes.
<code>noise.SNR_dB</code>	float	45	The signal-to-noise ratio in dB for the added white Gaussian noise, with 45 dB suggested by [37].

### Examples

```
>> data = daline.noise(data, 'noise.switchTrain', 1, 'noise.SNR_dB', 46);
```

```
>> data = daline.noise(data, 'data.fixRand', 1, 'noise.switchTest', 1, 'noise.SNR_dB', 45);
```

```
>> opt = daline.setopt('noise.switchTrain', 1, 'noise.switchTest', 1, 'noise.SNR_dB',  
    45);  
>> data = daline.noise(data, opt);
```

### Remark

- When `data.fixRand` is set to 1, not only is the randomness fixed, but each measurement within a sample — every data point in a row, for instance, within `data.train.Vm` — receives the same noise value. This suggests a scenario where the data for the entire system is captured by a single device at any given moment. However, this assumption may not accurately reflect real-world measurement practices. Also, such an addition of noise does not significantly impact the training performance. Hence, `data.fixRand` is 0 by default — in this case, each element within a row will be affected by a unique instance of noise. This implies that variables within the system are measured by different devices, which is a more realistic condition that has a greater influence on training performance.

### 3.2.3 Add Data Outliers (`daline.outlier`)

DALINE also includes the capability to inject outliers into datasets, simulating anomalies that are occasionally encountered in real-world data, especially in electrical measurements. This functionality is provided by the `daline.outlier` wrapper. The core idea behind `daline.outlier` is manipulating original data points by randomly applying multipliers defined by users to significantly amplify data values relative to the rest of the dataset. The primary and mandatory argument for `daline.outlier` is `data`, which must adhere to the data structure prescribed by DALINE. Examples of how to use this feature are demonstrated below, and the default, but adjustable, parameters for `daline.outlier` are outlined in Table 3.3.

### Examples

```
>> data = daline.outlier(data, 'outlier.switchTrain', 1, 'outlier.percentage', 2.5);
```

```
>> opt = daline.setopt('outlier.switchTrain', 1, 'outlier.switchTest', 1, 'outlier.  
    factor', 3);  
>> data = daline.outlier(data, opt);
```

Table 3.3: Default name-value input arguments for `daline.outlier`

Parameter	Format	Default	Description
<code>data.fixRand</code>	binary	0	Fixes randomness when introducing data outliers: 1 for yes, 0 for no.
<code>data.fixSeed</code>	integer	88	The seed used to fix the randomness in adding data outliers.
<code>outlier.switchTrain</code>	binary	1	Activates adding outliers to the training data: 0 for no, 1 for yes (1 by default when using <code>daline.outlier</code> , but 0 by default when using <code>daline.data</code> ).
<code>outlier.switchTest</code>	binary	0	Activates adding outliers to the testing data: 0 for no, 1 for yes.
<code>outlier.percentage</code>	float	5	The percentage of data points to be affected by outliers (here 5 means 5%), applied randomly to each variable. Care should be taken: even a small percentage can significantly impact the data set.
<code>outlier.factor</code>	float	2	The multiplier applied to the original measurements to generate outliers.

### Remark

- Take `outlier.percentage = 2.5` as an example. When `data.fixRand` is set to 1, not only is the randomness fixed, but it also means randomly selecting 2.5% of the rows within the training/testing dataset and amplifying their values simultaneously to simulate outliers (indicating a single-device data collection for the whole grid). When `data.fixRand` is set to 0 (default value), 2.5% of elements within each column of the training dataset will be randomly selected and multiplied to create outliers (indicating grid variables are measured by different devices); while 2.5% seems small, the cumulative effect across multiple columns significantly amplifies the likelihood of a row being classified as an outlier, due to the increased chance of encountering at least one amplified element per row.

### 3.2.4 Filter Data Noise (`daline.denoise`)

DALINE provides a tool for noise reduction in datasets through the `daline.denoise` wrapper, employing Kalman filtering. Kalman filtering is renowned for its effectiveness in filtering out noise from data. Again, the indispensable argument for `daline.denoise` is `data`, which requires agreement to the data format specified by DALINE. This section will provide examples to illustrate the usage of this feature. The limitations of `daline.denoise` are explained here as well. In addition, the modifiable default parameters of `daline.denoise` are detailed in Table 3.4.

### Examples

```
>> data = daline.denoise(data, 'filNoi.switchTrain', 1, 'filNoi.useARModel', false);
```

```
>> opt = daline.setopt('filNoi.switchTrain', 1, 'filNoi.useARModel', false, 'filNoi.  
    zeroInitial', 1);  
>> data = daline.denoise(data, opt);
```

### Remarks and limitations

- The configuration for the state transition matrix is crucial for noise filtering, as it reflects the system's evolution over time. The challenge lies in accurately capturing the dynamics and relationships between observations. Users can opt for a simple identity matrix by setting `filNoi.useARModel` to false, assuming independence among observations, or model the dynamics using an AutoRegressive (AR) model by setting `filNoi.useARModel` to true. However, each approach has its limitations in realistically representing time series data.
- Setting the process noise covariance, i.e., the parameter `filNoi.proNoiseLevel`, is a complex task due to its representation of unmodeled system dynamics or external disturbances, which are inherently uncertain and might not be directly observable. Manual tuning, though time-consuming, may lead to improved accuracy.
- The initialization of the estimated state is a delicate matter. While starting with a zero estimate is common, e.g., in [18, 38], it may not be accurate, particularly if the true initial state significantly deviates from zero. This can lead to a transient period where the filter adjusts to the actual state. Opting for the first observation as the initial estimate may mitigate this issue, which is also suggested here.

Table 3.4: Default name-value input arguments for `daline.denoise`.

Parameter	Format	Default	Description
<code>filNoi.switchTrain</code>	binary	1	Activates noise filtering for training data: 1 for yes (1 by default when using <code>daline.denoise</code> , but 0 by default when using <code>daline.data</code> ), 0 for no.
<code>filNoi.switchTest</code>	binary	0	Activates noise filtering for testing data: 1 for yes, 0 for no.
<code>filNoi.orderNum</code>	integer	5	The order number of the AutoRegressive model to estimate the data dynamics in the Kalman filter. A higher order allows for capturing more complex dynamics but increases computational complexity.
<code>filNoi.est_dB</code>	float	45	Represents the estimated signal noise level in decibels, which is converted to power to set the matrix of measurement noise covariance, influencing the filter's sensitivity to measurement noise.
<code>filNoi.useARModel</code>	boolean	false	Whether to use the AutoRegressive model for capturing data dynamics: true to use, false for an identity matrix as the transition matrix, assuming each observation evolves independently [38].
<code>filNoi.proNoiseLevel</code>	integer	100	Defines the magnitude of the matrix of process noise covariance, which models the inherent uncertainties or unmodeled dynamics in the system. A higher value indicates greater uncertainty in the system's evolution.
<code>filNoi.zeroInitial</code>	binary	0	Decides the initial state estimate in the Kalman filter. If set to 1, the initial estimate is zero, which may lead to a transient adjustment period. If 0, the initial estimate uses the first observation, potentially providing a closer starting point to the true state.

### 3.2.5 Filter Data Outliers (`daline.deoutlier`)

DALINE also provides a mechanism for outlier detection and removal through the `daline.deoutlier`



wrapper. The primary argument for `daline.deoutlier` is `data`, which must conform to the data format specified by DALINE. Below are examples to demonstrate the use of this wrapper. The default, yet adjustable, parameters for `daline.deoutlier` are presented in Table 3.5.

### Examples

```
>> data = daline.generate('case.name', 'case118', 'data.program', 'acpf', 'data.
    baseType', 'TimeSeriesRand');
>> data = daline.outlier(data, 'outlier.switchTrain', 1, 'outlier.factor', 2, '
    outlier.percentage', 0.5);
>> data = daline.deoutlier(data, 'filOut.switchTrain', 1, 'filOut.method', 'quartiles
    ', 'filOut.tol', 100);
```

```
>> data = daline.generate('case.name', 'case118', 'data.program', 'acpf', 'data.
    baseType', 'TimeSeriesRand');
>> data = daline.outlier(data, 'outlier.switchTrain', 1, 'outlier.factor', 2, '
    outlier.percentage', 0.5);
>> opt = daline.setopt('filOut.switchTrain', 1, 'filOut.method', 'quartiles');
>> data = daline.deoutlier(data, opt);
```

Table 3.5: Default name-value input arguments for `daline.deoutlier`.

Parameter	Format	Default	Description
<code>filOut.switchTrain</code>	binary	1	Activates outlier filtering for training data: 1 for yes (1 by default when using <code>daline.deoutlier</code> , but 0 by default when using <code>daline.data</code> ), 0 for no.
<code>filOut.switchTest</code>	binary	0	Activates outlier filtering for training data: 1 for yes, 0 for no.
<code>filOut.method</code>	string	'quartiles'	Specifies the method for outlier detection: options include 'median', 'mean', 'quartiles', and 'grubbs'.
<code>filOut.tol</code>	float	5	Sets the tolerance level for outlier detection. A higher value results in fewer detected outliers. This value must be between 0 and 1 when method is 'grubbs'.

### Remarks and limitations

- The functionality of `daline.deoutlier` is primarily built upon MATLAB's `isoutlier` function. This underlying approach ensures a robust and well-tested method for identifying and filtering outliers in the dataset.
- The parameter `filOut.method` allows users to select the criterion supported by `isoutlier` for outlier detection based on statistical measures. Each option corresponds to a different statistical method for defining what constitutes an outlier:
  - 'median': Outliers are identified based on their deviation from the median of the data.

- 'mean': Outliers are identified based on their deviation from the mean of the data.
- 'quartiles': Outliers are identified based on the interquartile range, which is the range between the first and third quartiles. This method is less affected by extreme values.
- 'grubbs': This method applies Grubbs' test for outliers, assuming the data is normally distributed.
- The `filOut.tol` parameter is used as the `ThresholdFactor` in the `isoutlier` function. It determines the sensitivity of the outlier detection process:
  - A higher `filOut.tol` value increases the threshold for detecting outliers, thus reducing the number of data points identified as outliers.
  - Conversely, a lower `filOut.tol` value makes the outlier detection more sensitive, potentially increasing the number of data points flagged as outliers.
- If the outliers in `data` are generated by `daline.outlier` without fixing the randomness, it's worth noting that even a 2.5% outlier rate can result in a substantial quantity of outliers due to their independent and accumulative impact across various columns, as mentioned earlier. These outliers can be detected and removed by `daline.outlier`, leaving only a small number (or even zero) of data points.

### 3.2.6 Normalize Data (`daline.normalize`)

DALINE offers a feature for data normalization through the `daline.normalize` wrapper, specifically employing unit energy normalization. This process ensures that each feature within the dataset has a consistent scale, which is crucial for many data processing and machine learning algorithms. The primary input for `daline.normalize` is `data`, which must be in the structured format as defined by DALINE. The following examples illustrate the application of this wrapper, and the configurable parameter for `daline.normalize` is detailed in Table 3.6.

#### Examples

```
>> data = daline.normalize(data);
```

#### Remarks

- Unit energy normalization is a process where each variable in the dataset is scaled such that the sum of the squares of its values equals one. This is achieved by dividing each value by the square root of the sum of the squares of all values in that variable. In other words, this normalization is performed independently for each variable, utilizing distinct scaling factors derived from their respective distributions. As a result, each variable is transformed to have unit energy, but the relative scales between different variables are altered.
- A consequence of applying unit energy normalization is that the original physical relationships among variables might not be preserved. The normalization changes the scale of each variable differently, which can affect the interpretation of these variables in relation to one another. Hence, the data normalized by `daline.normalize` may not be suitable for data-driven linearization methods that incorporate the physical knowledge of the power flow model. When calling these linearization methods in DALINE, users should use the original data.
- In datasets, the presence of columns comprised entirely of zeros (all-zero columns) can lead to these columns being transformed into NaN (Not a Number) following unit energy normalization. This

phenomenon arises because the normalization process involves division by a factor, which, for all-zero columns, is inherently zero, thus leading to undefined values. Situations that might result in all-zero columns include but are not limited to: instances where a bus maintains nodal active power balance over a certain duration or when the reference angle at the slack bus is designated as zero. To avoid this issue, users can tune the parameters `load.smallValue`, `gen.smallValue`, and `voltage.refAngle`, as outlined in Table 3.1, which are designed to eliminate all-zero columns. Nonetheless, it is crucial to bear in mind that the emergence of NaN values in normalized data warrants an initial examination for all-zero columns in the raw dataset.

Table 3.6: Default name-value input argument for `daline.normalize`.

Parameter	Format	Default	Description
<code>norm.switch</code>	binary	1	Activities normalization for both the training and testing datasets: 1 for yes (1 by default when using <code>daline.normalize</code> , but 0 by default when using <code>daline.data</code> ), 0 for no.

### 3.3 All-in-one Command for Data Generating/Processing (`daline.data`)

For users seeking a more streamlined approach than utilizing individual wrappers, DALINE offers `daline.data`, which consolidates all data generation and processing functionalities. Illustrated below are some examples.

#### Examples

```
>> data = daline.data('case.name', 'case118', 'num.trainSample', 500, 'num.testSample',
    300, 'data.program', 'acpf', 'data.baseType', 'TimeSeriesRand', 'noise.switchTrain',
    1, 'outlier.switchTrain', 1, 'norm.switch', 1);
```

```
>> opt = daline.setopt('case.name', 'case118', 'num.trainSample', 500, 'num.testSample',
    300, 'data.program', 'acpf', 'data.baseType', 'TimeSeriesRand', 'noise.switchTrain',
    1, 'outlier.switchTrain', 1, 'norm.switch', 1);
>> [data, X, Y] = daline.data(opt);
```

#### Remarks

- The configurable parameters for `daline.data` contain all those detailed in Tables 3.1 to 3.6.
- `daline.data` initiates by a mandatory process: generating synthetic data. Yet, other functionalities related to data processing can be turned off.
- When all functionalities are activated, the sequence of operations is as follows: data generation, addition of data outliers, addition of data noise, filtering of data outliers, filtering of data noise, and data normalization.
- Beyond the standard output `data`, `daline.data` also offers two optional outputs: `X` and `Y`, both of which are structured for ease of use. The `X` output contains data for all known variables, whereas

Y include data for all unknown variables. Their structures are explained below in detail. Note the numbers of PV buses, PQ buses, reference buses, and branches are defined as  $N_{PV}$ ,  $N_{PQ}$ ,  $N_{ref}$ , and  $N_{line}$ , respectively.

### Independent Variables for Training: **X.train**

- **X.train.P**:  $N_{train} \times (N_{PV} + N_{PQ})$  matrix, contains known active power injections at PV and PQ buses.
- **X.train.Q**:  $N_{train} \times N_{PQ}$  matrix, contains known reactive power injections at PQ buses.
- **X.train.Vm**:  $N_{train} \times (N_{PV} + N_{ref})$  matrix, contains known voltage magnitudes at PV and reference buses.
- **X.train.Vm2**:  $N_{train} \times (N_{PV} + N_{ref})$  matrix, contains known squared voltage magnitudes at PV and reference buses.
- **X.train.Va**:  $N_{train} \times N_{ref}$  matrix, contains the known voltage angle(s) at the reference bus(es).
- **X.train.PQ**:  $N_{train} \times (N_{PV} + 2 \times N_{PQ})$  matrix, contains all known active and reactive power injections at PV and PQ buses.
- **X.train.all**:  $N_{train} \times (2 \times (N_{PV} + N_{PQ}) + N_{ref})$  matrix, contains all known variables including active and reactive power injections, voltage magnitudes, and voltage angles.
- **X.train.all2**:  $N_{train} \times (2 \times (N_{PV} + N_{PQ}) + N_{ref})$  matrix, similar to **X.train.all** but with voltage magnitudes replaced by their squares.

### Independent Variables for Testing: **X.test**

- **X.test.P**:  $N_{test} \times (N_{PV} + N_{PQ})$  matrix, contains known active power injections at PV and PQ buses.
- **X.test.Q**:  $N_{test} \times N_{PQ}$  matrix, contains known reactive power injections at PQ buses.
- **X.test.Vm**:  $N_{test} \times (N_{PV} + N_{ref})$  matrix, contains known voltage magnitudes at PV and reference buses.
- **X.test.Vm2**:  $N_{test} \times (N_{PV} + N_{ref})$  matrix, contains known squared voltage magnitudes at PV and reference buses.
- **X.test.Va**:  $N_{test} \times N_{ref}$  matrix, contains the known voltage angle(s) at the reference bus(es).
- **X.test.PQ**:  $N_{test} \times (N_{PV} + 2 \times N_{PQ})$  matrix, contains all known active and reactive power injections at PV and PQ buses.
- **X.test.all**:  $N_{test} \times (2 \times (N_{PV} + N_{PQ}) + N_{ref})$  matrix, contains all known variables including active and reactive power injections, voltage magnitudes, and voltage angles.
- **X.test.all2**:  $N_{test} \times (2 \times (N_{PV} + N_{PQ}) + N_{ref})$  matrix, similar to **X.test.all** but with voltage magnitudes replaced by their squares.

### Dependent Variables for Training: **Y.train**

- **Y.train.Vm**:  $N_{train} \times N_{PQ}$  matrix, contains unknown voltage magnitudes at PQ buses.
- **Y.train.Vm2**:  $N_{train} \times N_{PQ}$  matrix, contains unknown squared voltage magnitudes at PQ buses.
- **Y.train.Va**:  $N_{train} \times (N_{PV} + N_{PQ})$  matrix, contains unknown voltage angles at PV and PQ buses.
- **Y.train.PF**:  $N_{train} \times N_{line}$  matrix, contains unknown active branch flows from the “From-bus”;

- **Y.train.PT**:  $N_{train} \times N_{line}$  matrix, contains unknown active branch flows from the “To-bus”;
- **Y.train.QF**:  $N_{train} \times N_{line}$  matrix, contains unknown reactive branch flows from the “From-bus”;
- **Y.train.QT**:  $N_{train} \times N_{line}$  matrix, contains unknown reactive branch flows from the “To-bus”;
- **Y.train.P**:  $N_{train} \times N_{ref}$  matrix, contains the unknown active power injection(s) at the reference bus(es).
- **Y.train.Q**:  $N_{train} \times (N_{PV} + N_{ref})$  matrix, contains unknown reactive power injection(s) at PV and reference buses.
- **Y.train.all**:  $N_{train} \times (2N_{PQ} + 2N_{PV} + 4N_{line} + 2N_{ref})$  matrix, contains all unknown variables, including voltage magnitudes, voltage angles, active and reactive power flows, and active and reactive power injections.
- **Y.train.all2**:  $N_{train} \times (2N_{PQ} + 2N_{PV} + 4N_{line} + 2N_{ref})$  matrix, similar to **Y.train.all** but with voltage magnitudes replaced by their squares.
- **Y.train.V**:  $N_{train} \times (2N_{PQ} + N_{PV})$  matrix, contains all the unknown voltage magnitudes and angles for PQ and PV buses.
- **Y.train.V2**:  $N_{train} \times (2N_{PQ} + N_{PV})$  matrix, similar to **Y.train.V** but includes squared voltage magnitudes for PQ buses.
- **Y.train.flow**:  $N_{train} \times 4N_{line}$  matrix, contains all the unknown active and reactive power flows in both directions.

### Dependent Variables for Testing: Y.test

- **Y.test.Vm**:  $N_{test} \times N_{PQ}$  matrix, contains unknown voltage magnitudes at PQ buses.
- **Y.test.Vm2**:  $N_{test} \times N_{PQ}$  matrix, contains unknown squared voltage magnitudes at PQ buses.
- **Y.test.Va**:  $N_{test} \times (N_{PV} + N_{PQ})$  matrix, contains unknown voltage angles at PV and PQ buses.
- **Y.test.PF**:  $N_{test} \times N_{line}$  matrix, contains unknown active branch flows from the “From-bus”;
- **Y.test.PT**:  $N_{test} \times N_{line}$  matrix, contains unknown active branch flows from the “To-bus”;
- **Y.test.QF**:  $N_{test} \times N_{line}$  matrix, contains unknown reactive branch flows from the “From-bus”;
- **Y.test.QT**:  $N_{test} \times N_{line}$  matrix, contains unknown reactive branch flows from the “To-bus”;
- **Y.test.P**:  $N_{test} \times N_{ref}$  matrix, contains the unknown active injection(s) at the reference bus(es).
- **Y.test.Q**:  $N_{test} \times (N_{PV} + N_{ref})$  matrix, contains unknown reactive power injection(s) at PV and reference buses.
- **Y.test.all**:  $N_{test} \times (2N_{PQ} + 2N_{PV} + 4N_{line} + 2N_{ref})$  matrix, contains all unknown variables, including voltage magnitudes, voltage angles, active and reactive power flows, and active and reactive power injections.
- **Y.test.all2**:  $N_{test} \times (2N_{PQ} + 2N_{PV} + 4N_{line} + 2N_{ref})$  matrix, similar to **Y.test.all** but with voltage magnitudes replaced by their squares.
- **Y.test.V**:  $N_{test} \times (2N_{PQ} + N_{PV})$  matrix, contains all the unknown voltage magnitudes and angles for PQ and PV buses.
- **Y.test.V2**:  $N_{test} \times (2N_{PQ} + N_{PV})$  matrix, similar to **Y.test.V** but includes squared voltage magnitudes for PQ buses.
- **Y.test.flow**:  $N_{test} \times 4N_{line}$  matrix, contains all the unknown active and reactive power flows in both directions.

## Chapter 4

# Model Fitting and Testing

DALINE aims to identify an optimal linear relationship between a set of predictors,  $\mathbf{X} \in \mathbb{R}^{N_s \times N_x}$ , and responses,  $\mathbf{Y} \in \mathbb{R}^{N_s \times N_y}$ , where each set comprises  $N_s$  measurements (i.e., `num.trainSample`). The relationship between them is parameterized by a coefficient matrix generally denoted by  $\beta$ . In DALINE, the linear relationships take one of the following forms

$$\textbf{Type 1: } \mathbf{Y} = \mathbf{X}\beta$$

$$\textbf{Type 2: } \mathbf{Y} = \mathbf{X}\beta(k), \quad k = 1 \cdots K$$

$$\textbf{Type 3: } \mathbf{Y} = \phi(\mathbf{X})\beta_\phi$$

**Type 1** is a single linear model with coefficient matrix  $\beta \in \mathbb{R}^{N_x \times N_y}$ . **Type 2** is a piecewise linear model, where  $\beta(k) \in \mathbb{R}^{N_x \times N_y}$  is the coefficient matrix for segment  $k$ , and  $K$  denotes the number of total segments. **Type 3**, parameterized by  $\beta_\phi \in \mathbb{R}^{N_\phi \times N_y}$ , describes the linear relationship between  $\mathbf{Y}$  and  $\phi(\mathbf{X})$ , where  $\phi: \mathbb{R}^{N_x} \rightarrow \mathbb{R}^{N_\phi}$  is a mapping function.

Each of the model fitting methods in DALINE uses a training dataset of known  $\mathbf{X}$  and  $\mathbf{Y}$  to get estimates of the coefficient matrix/matrices. These estimates are denoted in the following documentation with a “hat” sign  $\hat{\cdot}$ , e.g.,  $\hat{\beta}$  represents the estimated coefficient matrix of **Model 1**. By applying the estimated linear coefficients to a test dataset of  $\mathbf{X}$ , predictions of  $\mathbf{Y}$ , denoted by  $\hat{\mathbf{Y}}$ , can be derived.

All the supported methods and their abbreviations are listed in Table 2.1. The default model types for these methods, along with their generalizability and applicability to specific scenarios, are detailed in Table 4.1 (see our simulation work in [3] for the detailed analysis and discussion of Table 4.1). Note that users can change the type of the resulting models through settings, and any such changes will be reflected in the outputted model, as explained in Table 4.3.

Table 4.1: Default model types, generalizability and applicability of the linearization approaches

Approach	Goal	Model Type	Predictor Generalizability	Response Generalizability	Multicollinearity Applicability	Zero Predictor Applicability	Constant Predictor Applicability	Normalization Applicability
LS	Build	1	Arbitrary	Arbitrary	×	×	✓	✓
LS_SVD	Build	1	Arbitrary	Arbitrary	×	✓	✓	✓
LS_COD	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_HBLD	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_HBLE	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_TOL	Build	1	Arbitrary	Arbitrary	×	✓	✓	✓
LS_CLS	Build	1	Arbitrary	Arbitrary	×	×	✓	✓
LS_LIFX	Build	3	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_LIFXi	Build	3	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_WEI	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_REC	Update	1	Arbitrary	Arbitrary	×	×	✓	✓
LS_REP	Update	1	Arbitrary	Arbitrary	×	×	✓	✓
LS_PIN	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_PCA	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_GEN	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LS_HBW	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
PLS_SIM	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
PLS_SIMRX	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
PLS_BDOpen	Build	1	$P, Q$	$Vm, Va$	✓	×	×	✓
PLS_BDL	Build	1	$V, P, Q$	Arbitrary	✓	×	×	✓
PLS_BDLY2	Build	1	$V, P, Q$	Arbitrary	✓	×	×	✓
PLS_REC	Update	1	Arbitrary	Arbitrary	✓	✓	✓	✓
PLS_RECW	Update	1	Arbitrary	Arbitrary	✓	✓	✓	✓
RR	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
RR_VCS	Build	3	$V^2, P, Q$	$V^2, R_{ij}, C_{ij}$	✓	✓	✓	×
RR_KPC	Build	2	Arbitrary	Arbitrary	✓	✓	✓	✓
RR_WEI	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
SVR	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
SVR_CCP	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
SVR_POL	Build	3	Arbitrary	Arbitrary	✓	✓	✓	✓
SVR_RR	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LCP_BOX	Build	1	$P, Q$	$V, \theta$	✓	✓	✓	×
LCP_BOXN	Build	1	$P, Q$	$V, \theta$	✓	✓	✓	×
LCP_COU	Build	1	$V, \theta$	PF, PT, QF, QT	✓	✓	✓	×
LCP_COU2	Build	1	$V, \theta$	PF, PT, QF, QT	✓	✓	✓	×
LCP_COUN	Build	1	$V, \theta$	PF, PT, QF, QT	✓	✓	✓	×
LCP_COU2N	Build	1	$V, \theta$	PF, PT, QF, QT	✓	✓	✓	×
LCP_JGD	Build	1	$P, Q$	$V, \theta$	✓	✓	✓	×
LCP_JGDN	Build	1	$P, Q$	$V, \theta$	✓	✓	✓	×
DRC_XM	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
DRC_XYM	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
DRC_XYD	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
DC_LS	Build	1	$P$	$\theta$	✓	✓	✓	×
DLPF_C	Build	1	$V, \theta, P, Q$	$V, \theta, PF, QF$	✓	✓	✓	×
PLS_CLS	Build	2	Arbitrary	Arbitrary	✓	✓	✓	✓
PLS_NIP	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
DC	Build	1	$P$	$\theta$	✓	✓	✓	×
PTDF	Build	1	$P$	PF	✓	✓	✓	×
TAY	Build	1	$P, Q$	$V, \theta$	✓	✓	✓	×
DLPF	Build	1	$V, \theta, P, Q$	$V, \theta, PF, QF$	✓	✓	✓	×
PLS_REP	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
QR	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
LD	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
PIN	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
SVD	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
COD	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓
PCA	Build	1	Arbitrary	Arbitrary	✓	✓	✓	✓

**Terminology:** The term “build” refers to the method designed to construct a linear model from a static, historical dataset of electrical measurements. Conversely, “update” denotes the strategy to refine an existing linear model by progressively incorporating new measurements. Furthermore, models 1, 2, and 3 align with the model classifications outlined in Part I of this tutorial; refer to the Problem Formulation part in Section 1 of [2] for more details.

## 4.1 All-in-one Command for Model Fitting/Testing (`daline.fit`)

In DALINE, model fitting and testing is executed by running the `daline.fit` wrapper with a data struct containing the training and test datasets as the first argument (`data`). Again, `data` must adhere to DALINE's standardized data format, as detailed in Section 3.2.1 (the format of `data` will be automatically verified by `daline.fit`). Eventually, `daline.fit` returns the fitted model, together with model predictions, errors, and other relevant information, in the resulting `model` struct.

```
>> model = daline.fit(data);
```

Table 4.2 lists the possible input and output variables that  $\mathbf{X}$  and  $\mathbf{Y}$ , respectively, can contain, while Table 4.3 details the fields included in the `daline.fit` `model` object. Note that it is possible for bus-type variables to appear both in `model.predictorList` and `model.responseList`, since they refer to different buses in each.

Table 4.2: Possible predictor and response variables for DPFL models.

Predictors	Responses
1 (corresponding to the constant value in the linear model)	V <sub>m</sub> of PQ buses
V <sub>m</sub> of PV, slack buses	V <sub>m2</sub> of PQ buses
V <sub>m2</sub> of PV, slack buses	V <sub>a</sub> of PV, PQ buses
V <sub>a</sub> of slack bus	P of slack bus
P of PV, PQ buses	Q of slack bus
Q of PQ buses	PF of all branches
	PT of all branches
	QF of all branches
	QT of all branches

Following after the `data` positional argument, `daline.fit` also accepts optional name-value arguments. These can be passed to the function in any order, failing which, the default values are used. Some **general parameters** that are **not specific to linearization algorithms** are given in Table 4.4. Methods may have access to additional name-value options: if these are shared with other methods belonging to the same function class as them, they are listed at the start of each section of the said function class in the documentation below; if they are method-specific, they are listed under the **Additional inputs** subsection of that method.

```
>> model_A = daline.fit(data, 'method.name', 'RR', 'variable.predictor', {'P', 'Vm2'}, '
    RR.lambdaInterval', 0:1e-3:0.02);
% equivalent alternative
>> opt = daline.setopt('method.name', 'RR', 'variable.predictor', {'P', 'Vm2'}, 'RR.
    lambdaInterval', 0:1e-3:0.02);
>> model_B = daline.fit(data, opt);
```



Table 4.3: Internals of `daline.fit`'s outputted `model` object.

Name	Description
<code>algorithm</code>	Specifies the DALINE method that produced the model.
<code>predictorList</code>	A cell array of the predictors used in $\mathbf{X}$ .
<code>responseList</code>	A cell array of the responses contained in $\mathbf{Y}$ . Note that they refer only to the <i>direct</i> output of the DPFL model, and may not be identical to all the variables predicted by the method. For instance, the variables in <code>responseList</code> may require back-transformation to their original coordinates. In another example, DCPF only predicts $\mathbf{V_a}$ directly from linearisation. However, its model also returns results for $\mathbf{V_m}$ , $\mathbf{PF}$ , $\mathbf{PT}$ , $\mathbf{QF}$ , and $\mathbf{QT}$ .
<code>predictorIdx</code>	Lists the indices of the buses/branches from which each predictor variable is taken.
<code>responseIdx</code>	Lists the indices of the buses/branches for which each response variable is generated.
<code>Beta</code>	The $N_x \times N_y$ matrix $\hat{\beta}$ fit by the algorithm. In the case of piecewise models, <code>Beta</code> returns a cell array where each cell is an $N_x \times N_y$ matrix $\hat{\beta}(k)$ corresponding to a given linearized segment.
<code>error</code>	A struct containing several sub-fields. Each is a matrix with $N_s$ rows, collecting the relative errors for a type of predicted variable. The error of a single prediction $i$ is calculated as $\left  \frac{\hat{y}_i - y_i}{y_i} \right $ . Note that one of the sub-fields of <code>error</code> is <code>error.all</code> , which integrates the errors of all the predicted variables.
<code>yPrediction</code>	A struct containing several sub-fields. Each is a matrix with $N_s$ rows, collecting the predicted values of a type of response variable, as the testing result of the linear model using the testing dataset.
<code>yTrue</code>	A struct containing several sub-fields. Each is a matrix with $N_s$ rows, collecting the true values of a type of response variable in the test dataset.
<code>type</code>	Specifies the model type, where 1: single linear; 2: piecewise linear; 3: transformed linear.
<code>note</code>	Specifies if back-transformation from $\mathbf{V_m2}$ to $\mathbf{V_m}$ is performed. This happens when $\mathbf{V_m2}$ is included in <code>opt.variable.response</code> and <code>opt.variable.Vm22Vm</code> is set to 1. Most methods in DALINE support the <code>note</code> field.

Table 4.4: General name-value input arguments for `daline.fit`.

Parameter	Format	Default	Description
<code>method.name</code>	character	'QR'	The algorithm to be used for power flow linearization. Choose one from the following currently-supported methods: 'LS', 'QR', 'LD', 'LS_PIN', 'PIN', 'LS_SVD', 'SVD', 'LS_COD', 'COD', 'LS_PCA', 'PCA', 'LS_HBW', 'LS_HBLD', 'LS_HBLE', 'LS_GEN', 'LS_LIFX', 'LS_LIFXi', 'LS_TOL', 'LS_CLS', 'LS_WEI', 'LS_REC', 'LS_REP', 'PLS_SIM', 'PLS_SIMRX', 'PLS_NIP', 'PLS_BDL', 'PLS_BDLy2', 'PLS_BDLopen', 'PLS_REC', 'PLS_RECw', 'PLS_REP', 'PLS_CLS', 'RR', 'RR_WEI', 'RR_KPC', 'RR_VCS', 'SVR', 'SVR_POL', 'SVR_CCP', 'SVR_RR', 'LCP_BOXN', 'LCP_BOX', 'LCP_JGDN', 'LCP_JGD', 'LCP_COUN', 'LCP_COUN2', 'LCP_COU', 'LCP_COU2', 'DRC_XM', 'DRC_XYM', 'DRC_XYD', 'DC', 'DC_LS', 'DLPF', 'DLPF_C', 'PTDF', 'TAY'.
<code>warning.switch</code>	binary	1	1: Turn warnings off for all methods, including par-for loop workers; otherwise 0.
<code>variable.predictor</code>	cell array	{'P' 'Q', 'Vm2', 'Va'}	Selection of predictors, e.g., {'P', 'Q', 'Vm2'}, or {'P'}.
<code>variable.response</code>	cell array	{'Vm2', 'Va', 'PF', 'PT', 'QF', 'QT', 'P', 'Q'}	Selection of responses, e.g., {'PF', 'PT', 'QF', 'QT'}, or {'Vm'}.
<code>variable.Vm2Vm</code>	binary	1	1: Use <code>Vm2</code> for training, but show the error, prediction, and testing data of <code>Vm</code> instead of <code>Vm2</code> in the outputted <code>model</code> struct; otherwise 0.

Internally, `daline.fit` goes through the following steps:

1. Verifies the format of the inputted `data` struct according to the standard data format of DALINE.
2. Collects option settings.
3. Generates both the training and the test  $\mathbf{X}$  and  $\mathbf{Y}$  matrices from `data` according to the `variable.predictor` and `variable.response` arguments.
  - Adds an intercept column to the  $\mathbf{X}$ s and, if requested, lifts their dimensions as well (see 4.2.12 for details on dimension lifting).
4. Implements the method called by `daline.fit`.
  - Possibly carries out preliminary steps, such as further coordinate transformation of  $\mathbf{X}$ , or hyperparameter tuning.
  - Performs linearization to acquire  $\hat{\beta}$  (`model.Beta`).
5. Applies  $\hat{\beta}$  to the  $\mathbf{X}$  test data to get  $\hat{\mathbf{Y}}$ .
6. Calculates prediction errors.
7. Summarizes model results.

Instead of running the `daline.fit` wrapper, a method can also be run directly by appending the desired `method.name` to `func_algorithm_` and running the resulting function in a similar way as with `daline.fit`, except without the `method.name` name-value pair:

```
% linearizing using the least squares (LS) method  
>> model_A = func_algorithm_LS(data);  
% equivalent to  
>> model_B = daline.fit(data, 'method.name', 'LS');
```

Running the method functions as such skips the first two steps of the `daline.fit` process and is generally not preferred unless the user would like to make changes to the base code of these functions. Nonetheless, in the **Examples** subsections of each of the method descriptions in the following sections, they have been provided as example code over `daline.fit` for illustrative brevity.

The background information in the **More About** subsections for each of the methods also assumes basic knowledge of power flow linearisation and the classes of regression and optimization algorithms used. Therefore, it focuses more on explaining the specifics of how each algorithm was implemented in DALINE. For a comprehensive theoretical reexamination and numerical comparison of various linearization approaches, including their theories, capabilities, limitations, generalizability, applicability, actual accuracy, and computational efficiency, refer to [2]-[3]. Users can also refer to the references listed in each subsection for the technical details and examples of the linearization method demonstrated in that subsection.

Additionally, as outlined below, numerous methods are designed to leverage parallel computing for enhanced performance. When parallel computation is activated for the first time, MATLAB might take extra time to initialize its parallel computing toolkit (assuming it is already installed). This initial delay is worthwhile, as parallel computing significantly accelerates processing speeds (certainly, the actual gains depend on the number of CPU cores available).

## 4.2 Least Squares Family

This class of functions covers the following algorithms:

- Ordinary least squares (**LS**)
- Ordinary least squares with generalized inverse (**LS\_PIN**)
- Least squares with singular value decomposition (**LS\_SVD**)
- Least squares with complete orthogonal decomposition (**LS\_COD**)
- Least squares with principal component analysis (**LS\_PCA**)
- Least squares with Huber loss function: a direct solution (**LS\_HBLD**)
- Least squares with Huber loss function: an equivalent solution (**LS\_HBLE**)
- Least squares with Huber weighting function (**LS\_HBW**)
- Generalized least squares (**LS\_GEN**)
- Total least squares (**LS\_TOL**)
- Least squares with clustering (**LS\_CLS**)
- Least squares with lifting dimension and Moore-Penrose inverse (**LS\_LIFX**, **LS\_LIFXi**)
- Least squares with programming using weights for observations (**LS\_WEI**)
- Recursive least squares (**LS\_REC**)
- Repeated least squares (**LS\_REP**)

### 4.2.1 Ordinary Least Squares (LS)

#### Tips

- For this method to work, the predictor variables must all be linearly independent; i.e., the predictor matrix must have full column rank. Otherwise, this method cannot generate the desired linear power flow model owing to the data multicollinearity issue [2, 3].

#### Examples

```
>> model = daline.fit(data, 'method.name', 'LS');
```

```
>> model = daline.fit(data, 'method.name', 'LS', 'variable.predictor', {'P'}, 'variable.  
.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'LS', 'variable.predictor', {'P', 'Q'}, 'variable.  
.response', {'PF'});  
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS(data, 'variable.predictor', {'P', 'Q'}, 'variable.  
.response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});  
>> model = func_algorithm_LS(data, opt);
```

#### More About

This method finds the coefficient matrix  $\beta$  by minimizing the sum of squared residuals

$$\min_{\beta} \|Y - X\beta\|_2^2$$

which has the explicit solution

$$\hat{\beta} = (X^\top X)^{-1} X^\top Y$$

### 4.2.2 Ordinary Least Squares with Generalized Inverse (LS\_PIN)

#### Tips

- The generalized inverse can be applied even when  $X^\top X$  is singular. However, its use leads to inevitable computational error in networks with PV buses [39].
- Refer to Section 4.9.3 for a comparison of this method's performance with that of using the generalized inverse directly.

**Examples**

```
>> model = daline.fit(data, 'method.name', 'LS_PIN');
```

```
>> model = daline.fit(data, 'method.name', 'LS_PIN', 'variable.predictor', {'P'}, '
    variable.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'LS_PIN', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_PIN(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_LS_PIN(data, opt);
```

**More About**

The generalized inverse (a.k.a. the Moore-Penrose pseudoinverse) is used in this method to find the minimum  $L^2$  norm solution to a system of linear equations with infinite solutions. Specifically, the generalized inverse is implemented in

$$\hat{\beta} = \left( \mathbf{X}^\top \mathbf{X} \right)^{-1} \mathbf{X}^\top \mathbf{Y}$$

**4.2.3 Least Squares with Singular Value Decomposition (LS\_SVD)****Tips**

- The pseudoinverse can be applied even when  $\mathbf{X}^\top \mathbf{X}$  is singular. However, its use leads to inevitable computational error in networks with PV buses [39].
- Refer to Section 4.9.4 for a comparison of this method's performance with that of using the singular value decomposition directly.

**Examples**

```
>> model = daline.fit(data, 'method.name', 'LS_SVD');
```

```
>> model = daline.fit(data, 'method.name', 'LS_SVD', 'variable.predictor', {'P'}, '
    variable.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'LS_SVD', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_SVD(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_LS_SVD(data, opt);
```

### More About

This method factorizes the Gram matrix  $\mathbf{X}^\top \mathbf{X} \in \mathbb{R}^{N_x \times N_s}$  using the SVD  $\mathbf{X}^\top \mathbf{X} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^\top$ , where  $\mathbf{U}$  is an  $N_x$ -by- $N_x$  orthogonal matrix,  $\mathbf{\Sigma}$  is an  $N_x$ -by- $N_s$  matrix with singular values on its diagonal, and  $\mathbf{V}^\top$  is an  $N_s$ -by- $N_s$  orthogonal matrix. Its pseudoinverse is then implemented to find  $\hat{\beta}$  in

$$\hat{\beta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y} = (\mathbf{U} \mathbf{\Sigma} \mathbf{V}^\top)^{-1} \mathbf{X}^\top \mathbf{Y}$$

### References

Zhentong Shao et al. “Data Based Linear Power Flow Model: Investigation of a Least-Squares Based Approximation”. In: *IEEE Transactions on Power Systems* 36.5 (2021), pp. 4246–4258

## 4.2.4 Least Squares with Complete Orthogonal Decomposition (LS\_COD)

### Tips

- Complete orthogonal decomposition is a generalization of QR decomposition: when  $\mathbf{X}^\top \mathbf{X}$  is full rank,  $\mathbf{Z}$  becomes the identity matrix.
- Using complete orthogonal decomposition can not only address multicollinearity in the predictor matrix  $\mathbf{X}$ , but also speed up regression computations. This is because  $\tilde{\mathbf{R}}$  is an upper-triangular matrix with fewer dimensions, rendering  $\tilde{\mathbf{R}}^{-1}$  easier to compute.
- Refer to Section 4.9.5 for a comparison of this method’s performance with that of using the complete orthogonal decomposition directly.

### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_COD');
```

```
>> model = daline.fit(data, 'method.name', 'LS_COD', 'variable.predictor', {'P'}, '
    variable.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'LS_COD', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_COD(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_LS_COD(data, opt);
```

### More About

This method factorizes the Gram matrix  $\mathbf{X}^\top \mathbf{X} \in \mathbb{R}^{N_x \times N_s}$  using complete orthogonal decomposition to get

$$\mathbf{X}^\top \mathbf{X} = \tilde{\mathbf{Q}} \begin{bmatrix} \tilde{\mathbf{R}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{Z}^\top$$

where  $\tilde{\mathbf{Q}} \in \mathbb{R}^{N_x \times N_x}$  and  $\mathbf{Z} \in \mathbb{R}^{N_x \times N_x}$  are both orthogonal matrices, and  $\tilde{\mathbf{R}} \in \mathbb{R}^{N_{x_r} \times N_{x_r}}$  is an upper-triangular matrix. Its pseudoinverse is then implemented to find  $\hat{\beta}$  in

$$\hat{\beta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y} = \mathbf{Z} \begin{bmatrix} \tilde{\mathbf{R}}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \tilde{\mathbf{Q}}^\top \mathbf{X}^\top \mathbf{Y}$$

### References

Zhentong Shao et al. “Data Based Linear Power Flow Model: Investigation of a Least-Squares Based Approximation”. In: *IEEE Transactions on Power Systems* 36.5 (2021), pp. 4246–4258



## 4.2.5 Least Squares with Principal Component Analysis (LS\_PCA)

### Additional inputs

Table 4.5: Table of parameters specific to least squares with principal component analysis and direct principal component analysis.

Parameter	Format	Default	Description
PCA.parallel	binary	1	1: use parallel computation to speed up, otherwise 0
PCA.rank	binary	0	1: Use the rank of $\mathbf{X}$ as the number of components; 0: use PCA.PerComponent.
PCA.PerComponent	vector/float	[40:10:80]	Unit: %, i.e., the proportion of the number of principal components w.r.t. the number of predictors. If a vector is given, the toolbox will find the optimal percentage using cross-validation with parallelization; if a scalar is given, the inputted value will be used directly.
PCA.numFold	integer	10	The number of folds for cross-validation tuning of the number of principal components to be used. This number must be divisible by the number of training samples.
PCA.fixCV	binary	1	1: Fix the random seed for partitioning data in cross-validation; otherwise 0.
PCA.fixSeed	integer	88	Random seed number for cross-validation partitioning.

### Tips

- LS\_PCA is sensitive to variable scaling. For valid results, the predictor data are suggested to be normalized.
- Cross-validation is helpful in determining a better number of principal components because using more principal components does not guarantee a higher predictor accuracy and may lead to overfitting.
- When working with noisy data, the first few components found by PCA usually have a higher signal-to-noise ratio than later components, since they exhibit the greatest variance. In contrast, the last few principal components are often dominated by noise and can be dropped.
- The  $\hat{\beta}$  returned by LS\_PCA is not directly interpretable, as they are back-transformations of the  $\hat{\beta}^{PCA}$  that were found by regressing on the principal components.
- Refer to Section 4.9.6 for a comparison of this method's performance with that of using PCA directly.

### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_PCA');
```

```
>> model = daline.fit(data, 'method.name', 'LS_PCA', 'variable.predictor', {'P'}, '
    variable.response', {'PF', 'Vm'}, 'PCA.PerComponent', [30:10:90], 'PCA.numFold', 5)
    ;
```

```
>> opt = daline.setopt('method.name', 'LS_PCA', 'variable.predictor', {'P'}, 'variable.
    response', {'PF', 'Vm'}, 'PCA.PerComponent', [30:10:90], 'PCA.numFold', 5);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_PCA(data, 'variable.predictor', {'P'}, 'variable.
    response', {'PF', 'Vm'}, 'PCA.PerComponent', [30:10:90], 'PCA.numFold', 5);
```

```
>> opt = daline.setopt('variable.predictor', {'P'}, 'variable.response', {'PF', 'Vm'}, '
    PCA.PerComponent', [30:10:90], 'PCA.numFold', 5);
>> model = func_algorithm_LS_PCA(data, opt);
```

## More About

The underlying idea behind this method is to project the predictor matrix  $\mathbf{X}$  onto a new orthonormal basis; i.e., convert the features in the data into uncorrelated features (“principal components”) before training. The covariance of  $\mathbf{X}$  undergoes eigendecomposition as

$$\text{Cov}(\mathbf{X}) = \mathbf{D}\mathbf{\Lambda}\mathbf{D}^\top$$

where  $\mathbf{D} \in \mathbb{R}^{N_x \times N_x}$  consists of the eigenvectors of  $\text{Cov}(\mathbf{X})$ , and  $\mathbf{\Lambda}$  is a diagonal matrix of the corresponding eigenvalues. Then, the least squares solution of the principal component regression model

$$\min_{\boldsymbol{\beta}} \|\mathbf{Y} - \mathbf{X}\mathbf{D}\boldsymbol{\beta}^{PCA}\|_2^2$$

is given by

$$\hat{\boldsymbol{\beta}}^{PCA} = \left[ (\mathbf{X}\mathbf{D})^\top (\mathbf{X}\mathbf{D}) \right]^{-1} (\mathbf{X}\mathbf{D})^\top \mathbf{Y} \quad (4.1)$$

and the linear coefficients of the original predictor data are found as

$$\hat{\boldsymbol{\beta}} = \mathbf{D}\hat{\boldsymbol{\beta}}^{PCA}$$

## 4.2.6 Least Squares with Huber Loss Function: a Direct Solution (LS\_HBLD)

### Additional inputs

Table 4.6: Table of parameters specific to least squares with Huber loss function: a direct solution.

Parameter	Format	Default	Description
<code>HBL.parallel</code>	binary	1	1: Use parallel computation; otherwise 0.
<code>HBL.delta</code>	vector/float	0.02	The discrete range of the outlier tolerance, i.e., $\delta^{HUB}$ , for cross-validation, e.g., [0.02:0.002:0.03]. If a scalar is given, then use it directly without tuning.
<code>HBL.initialGuess</code>	float	0	The initial guess for all values of $\hat{\beta}$ . In the <code>LS_HBLD</code> subroutine, it is a scalar factor multiplied to a ones vector to supply the <code>x0</code> argument input when calling <code>fminunc</code> .
<code>HBL.directOptions</code>	function	<code>optimoptions('fminunc', 'Display', 'off', 'Algorithm', 'quasi-newton');</code>	The <code>options</code> argument input for calling <code>fminunc</code> to solve the regression problem ('fminunc' is built in the MATLAB Optimization Toolbox).
<code>HBL.numFold</code>	integer	5	The number of folds for cross-validation tuning of $\delta^{HUB}$ . This number must be divisible by the number of training samples.
<code>HBL.fixCV</code>	binary	1	1: Fix the random seed for partitioning data in cross-validation; otherwise 0.
<code>HBL.fixSeed</code>	integer	88	Random seed number for cross-validation partitioning.

### Tips

- The Huber loss intends to handle data outliers, as this method avoids squared-error loss' tendency to over-emphasize observations with large residuals during model fitting. The resulting regression problem can be solved efficiently as a convex program.
- Although the problem is convex, the quasi-Newton algorithm used by default in `fminunc` may not be able to find the solution. It is recommended that the problem be solved by converting it to an equivalent quadratic convex problem, as is done in `func_algorithm_LS_HBLE` in Section 4.2.7.
- Cross-validation for  $\delta^{HUB}$  is not implemented if `HBL.delta` is a scalar. As  $\delta^{HUB}$  functions as a threshold to distinguish between inliers and outliers, its tuning is recommended.
- It is highly recommended to enable parallel computing mode, as it can markedly enhance computational efficiency, particularly when the auto-tuning feature for hyperparameters is engaged.

### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_HBLD');
```

```
>> model = daline.fit(data, 'method.name', 'LS_HBLD', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'HBL.delta', [0.04:0.005:0.06]);
```

```
>> opt = daline.setopt('method.name', 'LS_HBLD', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'HBL.delta', [0.04:0.005:0.06]);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_HBLD(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'HBL.delta', [0.04:0.005:0.06]);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'HBL.delta', [0.04:0.005:0.06]);
>> model = func_algorithm_LS_HBLD(data, opt);
```

## More About

The Huber loss function modifies the regression loss function, such that

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{N_s} \sum_{j=1}^{N_y} H_i(\beta_j)$$

with

$$H_i(\beta_j) = \begin{cases} \varepsilon_{ij}^2, & \|\varepsilon_{ij}\|_1 \leq \delta^{HUB} \\ \delta^{HUB} (2\|\varepsilon_{ij}\|_1 - \delta^{HUB}), & \|\varepsilon_{ij}\|_1 > \delta^{HUB} \end{cases} \quad (4.2)$$

where  $\varepsilon_{ij} = y_{ij} - \mathbf{x}_i^\top \beta_j$ ,  $\|\cdot\|_1$  denotes the  $\ell^1$  norm, and  $\delta^{HUB} \in \mathbb{R}$  is a preset threshold.

## References

Yitong Liu, Zhengshuo Li, and Yu Zhou. “Data-Driven-Aided Linear Three-Phase Power Flow Model for Distribution Power Systems”. In: *IEEE Transactions on Power Systems* (2021)

## 4.2.7 Least Squares with Huber Loss Function: an Equivalent Solution (LS\_HBLE)

### Additional inputs

Table 4.7: Table of parameters specific to least squares with Huber loss function: an equivalent solution

Parameter	Format	Default	Description
HBL.parallel	binary	1	1: Use parallel computation; otherwise 0.
HBL.programType	character	'indivi'	'whole' puts all responses (i.e., the number of columns in $\mathbf{Y}$ ) into one optimization program to solve for all $\hat{\beta}$ at once; 'indivi' solves for $\hat{\beta}$ individually by building one optimization program for each response.
HBL.language	character	'yalmip'	Optimization toolbox to formulate the programming problem. Choose between 'cvx' or 'yalmip'.
HBL.solver	character	'fmincon'	Solver options; choose amongst 'fmincon', 'quadprog', 'Gurobi' ('quadprog' and 'fmincon' are built in the MATLAB Optimization Toolbox; for 'Gurobi', you need to install it manually).
HBL.cvxQuiet	binary	1	1: Suppress CVX output in the command window; otherwise 0.
HBL.yalDisplay	binary	0	1: Show YALMIP display; otherwise 0.
HBL.delta	vector/float	0.02	The discrete range of the outlier tolerance, i.e., $\delta^{HUB}$ , for cross-validation, e.g., [0.02:0.002:0.03]. If a scalar is given, then use it directly without tuning.
HBL.numFold	integer	5	The number of folds for cross-validation tuning of $\delta^{HUB}$ . This number must be divisible by the number of training samples.
HBL.fixCV	binary	1	1: Fix the random seed for partitioning data in cross-validation; otherwise 0.
HBL.fixSeed	integer	88	Random seed number for cross-validation partitioning.

### Tips

- It is recommended that **HBL.programType** be set to 'indivi', as the solution of an overall model usually requires long compute times and may cause MATLAB to hang. Naturally, the resultant optimal  $\hat{\beta}$  will also differ between the two approaches. Note that using the average error across dependent variables for evaluation of the algorithm may be unfair, since each dependent variable has been treated separately. A reasonable substitute could be to use the min/max error for each dependent variable as a performance indicator instead.
- Cross-validation for  $\delta^{HUB}$  is not implemented if **HBL.delta** is a scalar. As  $\delta^{HUB}$  functions as a threshold to distinguish between inliers and outliers, its tuning is recommended.
- It is highly recommended to use 'yalmip' as the language, 'fmincon' as the solver, and 'indivi' as the type of programming.
- It is highly recommended to enable parallel computing mode, as it can markedly enhance computational efficiency, particularly when the auto-tuning feature for hyperparameters is engaged.
- In the event that a program is interrupted while CVX is midway through solving, input the following codes into the MATLAB command window to shut down the CVX process. This will prevent future errors in calling/selecting the CVX solver, e.g., "The global CVX solver selection cannot be changed while a model is being constructed."

*% When having this error: "The global CVX solver selection cannot be changed while a model is being constructed," input the following codes into the command window.*

```
>> cvx_begin;
>> cvx_end;
```

## Examples

```
>> model = daline.fit(data, 'method.name', 'LS_HBLE');
```

```
>> model = daline.fit(data, 'method.name', 'LS_HBLE', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'HBL.language', 'yalmip', 'HBL.solver', 'fmincon',
    'HBL.delta', 0.01);
```

```
>> opt = daline.setopt('method.name', 'LS_HBLE', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'HBL.parallel', 0, 'HBL.language', 'yalmip', 'HBL.
    solver', 'fmincon');
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_HBLE(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'HBL.language', 'yalmip', 'HBL.solver', 'fmincon');
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'HBL.language', 'yalmip', 'HBL.solver', 'fmincon');
>> model = func_algorithm_LS_HBLE(data, opt);
```

## More About

The least squares problem with the Huber loss function can be converted to the equivalent quadratic convex problem

$$\begin{aligned}
 & \underset{\beta, \varepsilon, \mathbf{w}}{\text{minimize}} && \sum_{i=1}^{N_s} \sum_{j=1}^{N_y} \left( \frac{1}{2} \varepsilon_{ij}^2 + \delta^{HUB} w_{ij} \right) \\
 & \text{subject to} && -\varepsilon - \mathbf{w} \preceq \mathbf{Y} - \mathbf{X}\beta \preceq \varepsilon + \mathbf{w} \\
 & && 0 \preceq \varepsilon \preceq \delta^{HUB} \mathbf{1} \\
 & && \mathbf{w} \succeq 0
 \end{aligned}$$

where  $\varepsilon$ ,  $\mathbf{w}$ , and  $\mathbf{1}$  are  $N_s$ -by- $N_y$  matrices. This problem can be solved as a normal optimization problem using either the CVX or the YALMIP toolboxes in MATLAB. In the case where `HBL.programType` is set

to 'indivi', one column of  $\mathbf{Y}$ , instead of the entire matrix, is used in the above optimization formulation. Accordingly, the summation over  $j$  is taken out. The problem is solved  $N_y$  times, once for each dependent variable in  $\mathbf{Y}$ , and the resulting  $\hat{\beta}$  from each solution are concatenated column-wise.

## References

Yitong Liu, Zhengshuo Li, and Yu Zhou. "Data-Driven-Aided Linear Three-Phase Power Flow Model for Distribution Power Systems". In: *IEEE Transactions on Power Systems* (2021)

Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004

### 4.2.8 Least Squares with Huber Weighting Function (LS\_HBW)

#### Additional inputs

Table 4.8: Table of parameters specific to least squares with Huber weighting function.

Parameter	Format	Default	Description
HBW.parallel	binary	1	1: Use parallel computation; otherwise 0.
HBW.TuningConst	vector/float	[1:0.1:1.4]	The discrete range of $k^{HUB}$ for cross-validation, e.g., [1:0.1:1.4]. If a scalar is given, then use it directly without tuning.
HBW.PCA	binary	1	1: Conduct PCA on the predictor matrix; otherwise 0.
HBW.numComponentRatio	float	70	Used in PCA without cross-validation. Unit: %, i.e., the proportion of the number of principal components w.r.t. the number of predictors.
HBW.numFold	integer	10	The number of folds for cross-validation tuning of $k^{HUB}$ , the Huber tuning constant. This number must be divisible by the number of training samples.
HBW.fixCV	binary	1	1: Fix the random seed for partitioning data in cross-validation; otherwise 0.
HBW.fixSeed	integer	88	Random seed number for cross-validation partitioning.

#### Tips

- Cross-validation for the Huber tuning constant  $k^{HUB}$  is not implemented if HBW.TuningConst is a scalar. Smaller values of  $k^{HUB}$  defend against outliers better, but are less efficient when errors are normally distributed. Thus, tuning  $k^{HUB}$  is recommended. However, should a scalar argument be preferred, it can be set to a suggested value of 1.345, which gives 95% efficiency when errors are normal while still offering robustness against outliers.
- Applying PCA on the predictor matrix for dimensionality reduction can help to address data

collinearity and noise. However, its use does not guarantee improvement in predictor accuracy. In addition, for valid results, the predictor data must be normalized.

### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_HBW');
```

```
>> model = daline.fit(data, 'method.name', 'LS_HBW', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'HBW.TuningConst', [0.8:0.1:1.5], 'HBW.PCA', 0);
```

```
>> opt = daline.setopt('method.name', 'LS_HBW', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'HBW.TuningConst', [0.8:0.1:1.5], 'HBW.PCA', 0);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_HBW(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'HBW.TuningConst', [0.8:0.1:1.5], 'HBW.PCA', 0);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'HBW.TuningConst', [0.8:0.1:1.5], 'HBW.PCA', 0);
>> model = func_algorithm_LS_HBW(data, opt);
```

### More About

The least squares problem with the Huber loss function is equivalent to a robust regression problem that uses a Huber weight function:

$$\hat{\beta}^{(t)} = \underset{\beta}{\operatorname{argmin}} (\mathbf{Y} - \mathbf{X}\beta)^\top \mathbf{W}_{HUB}^{(t-1)} (\mathbf{Y} - \mathbf{X}\beta)$$

It has the explicit solution

$$\hat{\beta}^{(t)} = (\mathbf{X}^\top \mathbf{W}_{HUB}^{(t-1)} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{W}_{HUB}^{(t-1)} \mathbf{Y}$$

where  $\mathbf{W}_{HUB}^{(t-1)} = \operatorname{diag}\{\mathbf{w}^{(t-1)}\}$  is the current weight matrix, and

$$\mathbf{w}^{(t-1)} = \begin{cases} 1, & |\tilde{\mathbf{e}}^{(t-1)}| < k^{HUB} \\ \frac{k^{HUB}}{|\tilde{\mathbf{e}}^{(t-1)}|}, & |\tilde{\mathbf{e}}^{(t-1)}| \geq k^{HUB} \end{cases} \quad (4.3)$$

Here,  $k^{HUB}$  is the Huber tuning constant, and  $\tilde{\mathbf{e}}^{(t-1)} = \frac{\boldsymbol{\varepsilon}^{(t-1)}}{\hat{\sigma}\sqrt{1-h}}$  where  $\boldsymbol{\varepsilon}^{(t-1)} = \mathbf{Y} - \mathbf{X}\hat{\beta}^{(t-1)}$  is the matrix of residuals from the previous iteration,  $\hat{\sigma}$  is a robust measure of spread given by  $\hat{\sigma} = \text{MAR}/0.6745$ , where MAR refers to the mean absolute residual, and  $h$  is the vector comprising the diagonals of the hat matrix  $H = \mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ ; i.e., the leverage values from a least squares fit.



## 4.2.9 Generalized Least Squares (LS\_GEN)

### Additional inputs

Table 4.9: Table of parameters specific to generalized least squares.

Parameter	Format	Default	Description
LSG.parallel	binary	1	1: Use parallel computation; otherwise 0.
LSG.InnovMdl	character	'AR'	Model for the residuals' covariance, $\Omega$ . Options include 'AR', 'CLM', 'HC0'; see <a href="http://www.mathworks.com/help/econ/fpls.html">www.mathworks.com/help/econ/fpls.html</a> for all options and more details.

### Tips

- The primary application of this method arises when regression residuals are heteroscedastic and/or autocorrelated. If  $\hat{\Omega} = \mathbf{I}$ ,  $\hat{\beta}$  is returned as the ordinary least squares estimate.
- The true covariance matrix  $\Omega$  of  $\epsilon$  is typically unknown in real-world applications and challenging to estimate. Currently, arbitrary models for computing  $\Omega$  in this method are not enabled; choose from one of the models provided in [www.mathworks.com/help/econ/fpls.html](http://www.mathworks.com/help/econ/fpls.html).
- Using this method requires installation of the MATLAB Econometrics toolbox as the function calls on the toolbox's `fpls` script.
- In DPFL contexts, `fpls` often runs into errors in computing the inverse of the variable `R`, corresponding to the upper-triangular matrix  $\mathbf{R}$  mentioned below. To address this problem, the code of `fpls.m` can be modified as follows:

```
% change these lines in fpls.m
[Q,R] = qr(X,0);
coeff = R\ (Q'*y);

% to the following:
[Q,R] = qr(X,0);
if det(R) == 0
    coeff = pinv(R)*(Q'*y);
else
    coeff = R\ (Q'*y);
end
```

### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_GEN');
```

```
>> model = daline.fit(data, 'method.name', 'LS_GEN', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSG.InnovMdl', 'AR');
```

```
>> opt = daline.setopt('method.name', 'LS_GEN', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSG.InnovMdl', 'AR');
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_GEN(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSG.InnovMdl', 'AR');
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSG.InnovMdl', 'AR');
>> model = func_algorithm_LS_GEN(data, opt);
```

## More About

In this method, the covariance matrix of the residuals is defined as

$$\mathbb{V}ar(\boldsymbol{\varepsilon}|\mathbf{X}) = \sigma_{\varepsilon}^2 \boldsymbol{\Omega}$$

where  $\sigma_{\varepsilon}^2$  is a constant and  $\boldsymbol{\Omega} \in \mathbb{R}^{N_s \times N_s}$  is a positive definite symmetric matrix. This is in contrast to the usual ordinary least squares assumption that  $\mathbb{V}ar(\boldsymbol{\varepsilon}|\mathbf{X}) = \sigma_{\varepsilon}^2 \mathbf{I}$ .

The value of  $\boldsymbol{\Omega}$  is estimated based on a pre-specified model for the covariance of the residuals specified by `LSG.InnovMdl`. Since  $\boldsymbol{\Omega}$  is a function not only of the training data set, but also the model residuals  $\boldsymbol{\varepsilon}$ , an initial  $\hat{\boldsymbol{\beta}}$  to calculate  $\boldsymbol{\varepsilon}$  must be computed. This is done by performing ordinary least squares via the QR decomposition, as done in Section 4.9.1:

$$\hat{\boldsymbol{\beta}} = \mathbf{R}^{-1} \mathbf{Q}^{\top} \mathbf{Y}$$

If  $\mathbf{R}$  is singular, as is commonly the case in power system applications, then the generalized inverse of  $\mathbf{R}$  is taken above instead.

In the first full iteration of the algorithm, the solution of the generalized least squares method is found as

$$\hat{\boldsymbol{\beta}} = \left( \mathbf{X}^{\top} \boldsymbol{\Omega}^{-1} \mathbf{X} \right)^{-1} \mathbf{X}^{\top} \boldsymbol{\Omega}^{-1} \mathbf{Y}$$

to give a new estimate of  $\boldsymbol{\varepsilon}$  and thus also  $\boldsymbol{\Omega}$ . This process continues iteratively until  $\hat{\boldsymbol{\beta}}$  converges.

## References

The MathWorks Inc. *Feasible generalized least squares*. 2023. URL: <https://mathworks.com/help/econ/fgls.html#buicqm5-17>

Carl Mugnier et al. “Model-less/measurement-based computation of voltage sensitivities in unbalanced electrical distribution networks”. In: *2016 Power Systems Computation Conference (PSCC)*. IEEE. 2016, pp. 1–7

### 4.2.10 Total Least Squares (LS\_TOL)

#### Tips

- The primary application of this method arises when not only the dependent variables, but also the independent variables in the data have noise; i.e., total least squares can be robust to noise in both  $\mathbf{Y}$  and  $\mathbf{X}$ .
- This method assumes that the standard deviations of measurement noise are equal, which may not be true. To address this, a modification of the method, weighted total least squares, can be implemented instead. However, it involves an NP-hard problem that cannot be solved in polynomial time unless  $P = NP$ , and is thus only computationally feasible in small power systems.
- Technically, an exact solution is only possible when the submatrix  $\mathbf{V}_{yy}$  is non-singular and can be inverted. Given this, in `func_algorithm_LS_TOL`, the pseudoinverse of  $\mathbf{V}_{yy}$  is taken when  $\mathbf{V}_{yy}$  is not invertible.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_TOL');
```

```
>> model = daline.fit(data, 'method.name', 'LS_TOL', 'variable.predictor', {'P'}, '
    variable.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'LS_TOL', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_TOL(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_LS_TOL(data, opt);
```

#### More About

This method treats both  $\mathbf{X}$  and  $\mathbf{Y}$  as having noise. Given  $\mathbf{X}_0 \in \mathbb{R}^{N_s \times N_x}$  and  $\mathbf{Y}_0 \in \mathbb{R}^{N_s \times N_y}$ , which denote the ground truth realizations of  $\mathbf{x}$  and  $\mathbf{y}$ , we can thus express  $\mathbf{X} = \mathbf{X}_0 + \boldsymbol{\varepsilon}_x$  and  $\mathbf{Y} = \mathbf{Y}_0 + \boldsymbol{\varepsilon}_y$ .

Consequently, the problem

$$\begin{aligned} \min_{\boldsymbol{\beta}} \quad & \| [\Delta \mathbf{X} \ \Delta \mathbf{Y}] \|^2_2 \\ \text{s.t.} \quad & \mathbf{Y} + \Delta \mathbf{Y} = (\mathbf{X} + \Delta \mathbf{X}) \boldsymbol{\beta} \end{aligned} \tag{4.4}$$

is solved to minimize the orthogonal distance between the data point in  $[\mathbf{X} \ \mathbf{Y}]$  and the fitting hyperplane.

Using singular value decomposition where  $[\mathbf{X} \ \mathbf{Y}] = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ ,  $\mathbf{V} \in \mathbb{R}^{(N_x+N_y) \times (N_x+N_y)}$  is partitioned as

$$\mathbf{V} = \begin{bmatrix} \mathbf{V}_{xx} & \mathbf{V}_{xy} \\ \mathbf{V}_{yx} & \mathbf{V}_{yy} \end{bmatrix}$$

with  $\mathbf{V}_{xy} \in \mathbb{R}^{N_x \times N_y}$ . The solution is then given by

$$\hat{\boldsymbol{\beta}} = -\mathbf{V}_{xy} \mathbf{V}_{yy}^{-1}$$

## References

Yuxiao Liu et al. “A data-driven approach to linearize power flow equations considering measurement noise”. In: *IEEE Transactions on Smart Grid* 11.3 (2019), pp. 2576–2587

### 4.2.11 Least Squares with Clustering (LS\_CLS)

#### Additional inputs

Table 4.10: Table of parameters specific to least squares with clustering.

Parameter	Format	Default	Description
<code>LSC.parallel</code>	binary	1	1: Use parallel computation; otherwise 0.
<code>LSC.clusNumInterval</code>	vector/integer	[2:1:10]	The discrete range of the number of clusters to be tuned in cross-validation. If a scalar integer is given, then use it directly without tuning.
<code>LSC.cvNumFold</code>	integer	10	The number of folds for cross-validation tuning of the number of clusters to be used in K-means. This number must be divisible by the number of training samples.
<code>LSC.fixKmeans</code>	binary	1	1: Fix the random seed for K-means in <code>LS_CLS</code> for consistency in clustering; otherwise 0.
<code>LSC.fixCV</code>	binary	1	1: Fix the random seed for partitioning data in cross-validation; otherwise 0.
<code>LSC.fixSeed</code>	integer	88	Random seed number for K-means and cross-validation partitioning.

#### Tips

- The primary application of this method arises when separate linear power flow models can be fit from different power system operating modes. Due to frequent ambiguity in the cross-validation-determined optimal number of clusters to use in regression, this method works best when the number

of operating modes, and thus the number of clusters to be used, is already known.

- In large power systems with many independent variables, using a large number of clusters can easily lead to underdetermined linear systems; i.e., there may be fewer than  $N_y$  observations belonging to a cluster. This leads to invalid results from LS\_CLS, or in a more general sense, from all DPFL methods that integrate clustering.
- For this method to work, the predictor variables must all be linearly independent; i.e., the predictor matrix must have full column rank. Otherwise, this method cannot generate the desired linear power flow model owing to the data multicollinearity issue [2, 3].

## Examples

```
>> model = daline.fit(data, 'method.name', 'LS_CLS');
```

```
>> model = daline.fit(data, 'method.name', 'LS_CLS', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSC.fixKmeans', 0, 'LSC.clusNumInterval', [6:2:12], 'LSC.cvNumFold', 5);
```

```
>> opt = daline.setopt('method.name', 'LS_CLS', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSC.fixKmeans', 0, 'LSC.clusNumInterval', [6:2:12], 'LSC.cvNumFold', 5);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_CLS(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSC.fixKmeans', 0, 'LSC.clusNumInterval', [6:2:12], 'LSC.cvNumFold', 5);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSC.fixKmeans', 0, 'LSC.clusNumInterval', [6:2:12], 'LSC.cvNumFold', 5);
>> model = func_algorithm_LS_CLS(data, opt);
```

## More About

$\mathbf{X}$  is first divided into  $K$  clusters using K-means clustering. Ordinary least squares is subsequently implemented on the partitioned data from each of the  $K$  clusters, such that the regression coefficients of the  $k$ -th segment are found as

$$\hat{\beta}(k) = \left[ \mathbf{X}(k)^\top \mathbf{X}(k) \right]^{-1} \mathbf{X}(k)^\top \mathbf{Y}(k), \forall k$$

To apply the resulting piecewise model, identify the cluster that a given input  $\mathbf{x}$  belongs to via

$$k = \arg \min_j \|\mathbf{x} - \boldsymbol{\mu}(j)\|_2^2$$

where  $\mu(j)$  is the centroid of cluster  $j$ . Then, use the corresponding  $\hat{\beta}$  to generate the prediction of  $\mathbf{y}$  as  $\hat{\mathbf{y}} = \hat{\beta}(k)^\top \mathbf{x}$ .

### 4.2.12 Least Squares with Lifting Dimension: Lifting the Whole $\mathbf{x}$ Jointly (LS\_LIFX)

#### Additional inputs

Table 4.11: Table of parameters specific to least squares with lifting dimension: lifting the whole  $\mathbf{x}$  jointly.

Parameter	Format	Default	Description
<code>variable.lift</code>	binary	1	It must be 1, i.e., the dimension lifting mode must be turned on.
<code>variable.liftX</code>	binary	1	It must be 1, i.e., it must perform the lifting defined in Korda and Mezić [17].
<code>variable.liftFixC</code>	binary	1	1: Fix the randomness of dimension lifting, i.e., the value of $\mathbf{c}$ as users will see below; otherwise 0.
<code>variable.liftNumDim</code>	vector/integer	[ ]	The number of lifted dimensions, $N_d$ . The default input of [ ] equates this number to the dimension of the predictor matrix $\mathbf{X}$ .
<code>variable.liftEps</code>	float	1	Hyperparameter $\varepsilon$ in the 'gauss' and 'invquad' lifting functions.
<code>variable.liftK</code>	float	1	Hyperparameter $K$ in the 'polyharmonic' lifting function.
<code>variable.liftType</code>	character	'gauss'	Lifting function used. Choose amongst 'gauss', 'invquad', 'invquad_ref', 'invmultquad', 'invmultquad_ref', 'polyharmonic', 'polyharmonic_ref', 'thinplate'.

#### Tips

- `variable.liftNumDim` must take an integer scalar value from 0 to  $N_x$ .
  - A value of 0 means that none of the variables in  $\mathbf{X}$  are lifted, such that the returned results are identical to those from LS\_PIN.
  - A value  $n$  between 0 to  $N_x$  will lift the first  $n$  variables in  $\mathbf{X}$ .
  - Argument values larger than  $N_x$  will automatically be reverted to take the value of  $N_x$ .
- The elementary function  $f_b$  is computed according to line 24 in the open-source code file “`ref.m`” from Korda and Mezić [17], i.e., lifting the whole  $\mathbf{x}$  simultaneously.
- Considering that increasing the lifting dimensions substantially enlarges the training dataset size, for exceptionally large systems (e.g., over 1000 buses), this expansion could cause the dataset to surpass MATLAB’s memory capacity.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_LIFX');
```

```
>> model = daline.fit(data, 'method.name', 'LS_LIFX', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'variable.liftType', 'polyharmonic', 'variable.
    liftK', 2);
```

```
>> opt = daline.setopt('method.name', 'LS_LIFX', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'variable.liftType', 'polyharmonic', 'variable.
    liftK', 2);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_LIFX(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'variable.liftType', 'polyharmonic', 'variable.liftK', 2);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'variable.liftType', 'polyharmonic', 'variable.liftK', 2);
>> model = func_algorithm_LS_LIFX(data, opt);
```

## More About

This method finds the coefficient matrix  $\beta_{\text{lift}}$  by performing dimension lifting and minimizing the sum of squared residuals

$$\min_{\beta_{\text{lift}}} \|\mathbf{Y} - \mathbf{X}_{\text{lift}}\beta_{\text{lift}}\|_2^2$$

which has the explicit solution

$$\hat{\beta}_{\text{lift}} = \left( \mathbf{X}_{\text{lift}}^\top \mathbf{X}_{\text{lift}} \right)^{-1} \mathbf{X}_{\text{lift}}^\top \mathbf{Y}$$

Here, the generalized inverse (a.k.a. the Moore-Penrose pseudoinverse) is used to find  $(\mathbf{X}_{\text{lift}}^\top \mathbf{X}_{\text{lift}})^{-1}$ .

Each row of  $\mathbf{X}_{\text{lift}} \in \mathbb{R}^{N_s \times (N_x + N_d)}$  is of the form

$$\mathbf{x}_{\text{lift}} = \begin{bmatrix} \mathbf{x} \\ \psi(\mathbf{x}) \end{bmatrix}^\top$$

and comprises original observation  $\mathbf{x}$  and  $N_D$  lift dimension functions  $\boldsymbol{\psi}(\mathbf{x})$  defined as

$$\boldsymbol{\psi}(\mathbf{x}) = \begin{bmatrix} \psi_1(\mathbf{x}) \\ \vdots \\ \psi_D(\mathbf{x}) \end{bmatrix}$$

where  $\psi_i(\mathbf{x}) = f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i)$ .  $\mathbf{c}_i$  is a  $\mathbb{R}^{N_x \times 1}$  base vector for the  $i^{\text{th}}$  lifted dimension, and takes a random value from the range of each independent variable in  $\mathbf{X}$ . Lift dimension functions which are currently supported in DALINE are listed in Table 4.12. Note that the elementary function  $f_b$  in Table 4.12 is given by  $f_b(\mathbf{x} - \mathbf{c}_i) = \sum_{j=1}^{N_x} (\mathbf{x} - \mathbf{c}_{ij})^2$  in the method LS\_LIFX when `variable.liftX` = 1

Table 4.12: Table of lifting functions usable as arguments to `variable.liftType` in `func_algorithm_LS_LIF`. Functions with names with a '-ref' suffix refer to those taken from Guo et al. [39].

Function	Description
'gauss'	$f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i) = e^{-\varepsilon^2 \cdot f_b(\mathbf{x} - \mathbf{c}_i)}$
'invquad'	$f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i) = 1/(1 + \varepsilon^2 \cdot f_b(\mathbf{x} - \mathbf{c}_i))$
'invquad_ref'	$f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i) = 1/(1 + e^{f_b(\mathbf{x} - \mathbf{c}_i)})$
'invmultquad'	$f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i) = 1/\sqrt{1 + \varepsilon^2 \cdot f_b(\mathbf{x} - \mathbf{c}_i)}$
'invmultquad_ref'	$f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i) = 1/\sqrt{1 + e^{f_b(\mathbf{x} - \mathbf{c}_i)}}$
'polyharmonic'	$f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i) = (f_b(\mathbf{x} - \mathbf{c}_i))^{K/2} \log \sqrt{f_b(\mathbf{x} - \mathbf{c}_i)}$
'polyharmonic_ref'	$f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i) = \sqrt{f_b(\mathbf{x} - \mathbf{c}_i)} \log \sqrt{f_b(\mathbf{x} - \mathbf{c}_i)}$
'thinplate'	$f_{\text{lift}}(\mathbf{x} - \mathbf{c}_i) = f_b(\mathbf{x} - \mathbf{c}_i) \log \sqrt{f_b(\mathbf{x} - \mathbf{c}_i)}$

## References

- Li Guo et al. "Data-driven Power Flow Calculation Method: A Lifting Dimension Linear Regression Approach". In: *IEEE Transactions on Power Systems* (2021)
- Milan Korda and Igor Mezić. "Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control". In: *Automatica* 93 (2018), pp. 149–160



### 4.2.13 Least Squares with Lifting Dimension: Lifting the Elements of $\mathbf{x}$ Individually (LS\_LIFXi)

#### Additional inputs

Table 4.13: Table of parameters specific to least squares with lifting dimension: lifting the elements of  $\mathbf{x}$  individually.

Parameter	Format	Default	Description
<code>variable.lift</code>	binary	1	It must be 1, i.e., the dimension lifting mode must be turned on.
<code>variable.liftX</code>	binary	0	It must be 0, i.e., it must perform the lifting defined in Guo et al. [16].
<code>variable.liftFixC</code>	binary	1	1: Fix the randomness of dimension lifting, i.e., the value of $\mathbf{c}$ as users will see below; otherwise 0.
<code>variable.liftNumDim</code>	vector/integer	[ ]	The number of lifted dimensions, $N_d$ . The default input of [ ] equates this number to the dimension of the predictor matrix $\mathbf{X}$ .
<code>variable.liftEps</code>	float	1	Hyperparameter $\varepsilon$ in the 'gauss' and 'invquad' lifting functions.
<code>variable.liftK</code>	float	1	Hyperparameter $K$ in the 'polyharmonic' lifting function.
<code>variable.liftType</code>	character	'gauss'	Lifting function used. Choose amongst 'gauss', 'invquad', 'invquad_ref', 'invmultquad', 'invmultquad_ref', 'polyharmonic', 'polyharmonic_ref', 'thinplate'.

#### Tips

- See the general tips described in Section 4.2.12.
- The elementary function  $f_b$  is computed as given by Equation (12) in Guo et al. [16], i.e., lifting the elements of  $\mathbf{x}$  individually.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_LIFXi');
```

```
>> model = daline.fit(data, 'method.name', 'LS_LIFXi', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'variable.liftType', 'polyharmonic', 'variable.
    liftK', 2);
```

```
>> opt = daline.setopt('method.name', 'LS_LIFXi', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'variable.liftType', 'polyharmonic', 'variable.
    liftK', 2);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_LIFXi(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'variable.liftType', 'polyharmonic', 'variable.liftK', 2);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'variable.liftType', 'polyharmonic', 'variable.liftK', 2);
>> model = func_algorithm_LS_LIFXi(data, opt);
```

## More About

The content here is mostly the same as the “**More About**” in Section 4.2.12. The only difference is that, in the method LS\_LIFXi when `variable.liftX = 0`, the elementary function  $f_b$  mentioned in Table 4.12 is given by  $f_b(\mathbf{x} - \mathbf{c}_i) = \sum_{j=1}^{N_x} (x_i - c_{ij})^2$ , where  $x_i$  is the  $i^{\text{th}}$  element in  $\mathbf{x}$ .

## References

- Li Guo et al. “Data-driven Power Flow Calculation Method: A Lifting Dimension Linear Regression Approach”. In: *IEEE Transactions on Power Systems* (2021)
- Milan Korda and Igor Mezić. “Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control”. In: *Automatica* 93 (2018), pp. 149–160

### 4.2.14 Weighed Least Squares (LS\_WEI)

#### Additional inputs

Table 4.14: Table of parameters specific to the weighted least squares method.

Parameter	Format	Default	Description
<code>LSW.omega</code>	float	0.6	Weight value $\omega$ with a valid range from 0 to 1. The default value of 0.6 is from [9].
<code>LSW.programType</code>	character	'whole'	'whole' puts all responses (i.e., the number of columns in $\mathbf{Y}$ ) into one optimization program to solve for all $\hat{\beta}$ at once; 'indivi' solves for $\hat{\beta}$ individually by building one optimization program for each response.
<code>LSW.language</code>	character	'cvx'	Optimization toolbox to formulate the programming problem. Choose between 'cvx' or 'yalmip'.
<code>LSW.solver</code>	character	'SeDuMi'	Solver options, e.g., 'quadprog', 'Gurobi', 'SDPT3', 'SeDuMi' ('SDPT3' and 'SeDuMi' are included in DALINE via CVX; 'quadprog' and 'fmincon' are built in the MATLAB Optimization Toolbox; for 'Gurobi', you need to install it manually.).
<code>LSW.parallel</code>	binary	1	1: Use parallel computation; otherwise 0. Only valid when <code>LSW.language</code> = 'yalmip' and <code>LSW.programType</code> = 'indivi', because to the best of the authors' knowledge, <code>cvx</code> does not support parallel computing.
<code>LSW.cvxQuiet</code>	binary	1	1: Suppress CVX output in the command window; otherwise 0.
<code>LSW.yalDisplay</code>	binary	0	1: Show YALMIP display; otherwise 0.

#### Tips

- The primary application of this method arises when data is collected sequentially in time, at regular intervals. The inputs  $\mathbf{X}$  and  $\mathbf{Y}$  given to this method must be ordered such that their first row comprises the oldest observation, and their last row the newest observation.
- If `LSW.omega` is set to a value less than 1, the weights of earlier measurements decrease exponentially with the number of updates. Although this helps in situations where the system operating point is changing, over time, it also causes earlier observations to take on practically insignificant weights. Thus, to optimize prediction accuracy, the use of cross-validation for the forgetting factor `LSW.omega` is recommended.
- It is recommended to use 'cvx' as the language, 'SeDuMi' as the solver, and 'whole' as the type of programming.
- In the event that a program is interrupted while CVX is midway through solving, input the following codes into the MATLAB command window to shut down the CVX process. This will prevent future errors in calling/selecting the CVX solver, e.g., "The global CVX solver selection cannot be changed while a model is being constructed."

*% When having this error: "The global CVX solver selection cannot be changed while a model is being constructed," input the following codes into the command window.*

```
>> cvx_begin;
>> cvx_end;
```

## Examples

```
>> model = daline.fit(data, 'method.name', 'LS_WEI');
```

```
>> model = daline.fit(data, 'method.name', 'LS_WEI', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSW.omega', 0.95, 'LSW.programType', 'whole', 'LSW.solver', 'SeDuMi', 'LSW.language', 'cvx', 'LSW.programType', 'whole');
```

```
>> opt = daline.setopt('method.name', 'LS_WEI', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSW.omega', 0.95, 'LSW.programType', 'whole', 'LSW.solver', 'SeDuMi');
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_WEI(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSW.omega', 0.95, 'LSW.programType', 'whole', 'LSW.solver', 'SeDuMi');
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSW.omega', 0.95, 'LSW.programType', 'whole', 'LSW.solver', 'SeDuMi');
>> model = func_algorithm_LS_WEI(data, opt);
```

## More About

This method employs a forgetting mechanism that uses the forgetting factor  $\omega$  to weight data points that are farther away in time from the target operating point less heavily. Consequently, the objective function becomes

$$\min_{\beta} \|\Omega(\mathbf{Y} - \mathbf{X}\beta)\|_2^2$$

where  $\Omega$  is a  $N_s$ -by- $N_y$  weight matrix, with each element in its  $i^{\text{th}}$  row taking the value  $\omega^{N_s-i}$ . This implies that only the most recent observation  $\mathbf{x}_{N_s}$  in the predictor matrix  $\mathbf{X}$  is fully weighted. Note that the minimization problem does not have a solution that can be computed analytically as  $\Omega$  is singular and cannot be inverted.

In the case where `LSW.programType` is set to 'indivi', one column of  $\mathbf{Y}$ , instead of the entire matrix, is used in the above optimization formulation. The problem is solved  $N_y$  times, once for each dependent variable in  $\mathbf{Y}$ , and the resulting  $\hat{\beta}$  from each solution are concatenated column-wise.

## References

Hanchen Xu et al. “Data-driven voltage regulation in radial power distribution systems”. In: *IEEE Transactions on Power Systems* 35.3 (2019), pp. 2133–2143

### 4.2.15 Recursive Least Squares (LS\_REC)

#### Additional inputs

Table 4.15: Table of parameters specific to recursive least squares.

Parameter	Format	Default	Description
LSR.recursivePercentage	float	40	Percentage of the training data set that comprises new data. Each new observation will be learnt recursively.
LSR.initializeP	binary	1	1: Initialize $\mathbf{P}$ with a large value; otherwise 0.
LSR.largeValueP	float	1e6	Large value for initializing $\mathbf{P}$ .
LSR.forgetFactor	float	0.99	Forgetting factor $\lambda$ ; generally takes values between 0.95 and 0.99.

#### Tips

- The primary application of this method arises in non-stationary environments where the underlying data-generating process may change over time. Smaller values of  $\lambda$  lend more weight to recent observations, increasing the tracking ability of the algorithm for new data. However, this may also increase sensitivity to noise. Conversely,  $\lambda$  closer to 1 allows for faster convergence.
- When  $\lambda = 1$ , this method is equivalent to applying ordinary least squares repeatedly to integrate new observations.
- For this method to work, especially when  $\lambda = 1$ , the predictor variables must all be linearly independent; i.e., the predictor matrix must have full column rank. Otherwise, this method cannot generate the desired linear power flow model owing to the data multicollinearity issue [2, 3].

#### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_REC');
```

```
>> model = daline.fit(data, 'method.name', 'LS_REC', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'LSR.recursivePercentage', 30, 'LSR.initializeP',
    0);
```

```
>> opt = daline.setopt('method.name', 'LS_REC', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'LSR.recursivePercentage', 30, 'LSR.initializeP',
```

```
0);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_REC(data, 'variable.predictor', {'P', 'Q'}, 'variable.
response', {'PF', 'Vm'}, 'LSR.recursivePercentage', 30, 'LSR.initializeP', 0);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'},
'LSR.recursivePercentage', 30, 'LSR.initializeP', 0);
>> model = func_algorithm_LS_REC(data, opt);
```

## More About

This method is an adaptive algorithm that updates regression coefficients as new data becomes available. The old measurements of  $\mathbf{x}$  and  $\mathbf{y}$  from time step 1 to  $t$  are denoted by  $\mathbf{X}[t] \in \mathbb{R}^{t \times N_x}$  and  $\mathbf{Y}[t] \in \mathbb{R}^{t \times N_y}$  respectively. Furthermore,  $\beta[t]$  refers to the coefficients derived from them using ordinary least squares at time step  $t$ .

At time step  $t + 1$ , the recursive least squares algorithm updates the coefficient vector  $\beta$  with the new measurements  $\mathbf{x}_{t+1}$  and  $\mathbf{y}_{t+1}$  via

$$\beta[t + 1] = \beta[t] + \mathbf{K}[t + 1](\mathbf{y}_{t+1} - \beta[t]^\top \mathbf{x}_{t+1})$$

where  $\mathbf{K}[t + 1] \in \mathbb{R}^{N_x \times 1}$  is the gain vector, computed as

$$\mathbf{K}[t + 1] = \frac{\mathbf{P}[t]\mathbf{x}_{t+1}}{\lambda + \mathbf{x}_{t+1}^\top \mathbf{P}[t]\mathbf{x}_{t+1}}$$

Here,  $\mathbf{P}[t] \in \mathbb{R}^{N_x \times N_x}$  is the inverse of the covariance matrix of  $\mathbf{X}[t]$  if `LSR.initializeP` is set to 0. Otherwise, it is a diagonal matrix taking the value of `LSR.largeValueP` along its diagonal.  $\mathbf{P}[t]$  can be updated by

$$\mathbf{P}[t + 1] = \frac{1}{\lambda} \left( \mathbf{P}[t] - \mathbf{K}[t + 1]\mathbf{x}_{t+1}^\top \mathbf{P}[t] \right)$$

Parameter  $\lambda \in \mathbb{R}$  is a forgetting factor between 0 and 1.

## References

- Yitong Liu, Zhengshuo Li, and Shumin Sun. “A Data-Driven Method for Online Constructing Linear Power Flow Model”. In: *IEEE Transactions on Industry Applications* (2023)
- Manoj Badoni, Alka Singh, and Bhim Singh. “Variable Forgetting Factor Recursive Least Square Control Algorithm for DSTATCOM”. in: *IEEE Transactions on Power Delivery* 30.5 (2015), pp. 2353–2361. DOI: [10.1109/TPWRD.2015.2422139](https://doi.org/10.1109/TPWRD.2015.2422139)

## 4.2.16 Repeated Least Squares (LS\_REP)

### Additional inputs

Table 4.16: Table of parameters specific to repeated least squares.

Parameter	Format	Default	Description
LSR.recursivePercentage	float	40	Percentage of the training data set that comprises new data. Each new observation will be learnt recursively.

### Tips

- This function is intended as a baseline for comparison with LS\_REC in terms of accuracy and computational efficiency.
- For this method to work, the predictor variables must all be linearly independent; i.e., the predictor matrix must have full column rank. Otherwise, this method cannot generate the desired linear power flow model owing to the data multicollinearity issue [2, 3].

### Examples

```
>> model = daline.fit(data, 'method.name', 'LS_REP');
```

```
>> model = daline.fit(data, 'method.name', 'LS_REP', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSR.recursivePercentage', 30);
```

```
>> opt = daline.setopt('method.name', 'LS_REP', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSR.recursivePercentage', 30);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LS_REP(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSR.recursivePercentage', 30);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'LSR.recursivePercentage', 30);
>> model = func_algorithm_LS_REP(data, opt);
```

### More About

This method is essentially recursive least squares, but with the forgetting factor  $\lambda$  fixed to have the value of 1. This amounts to repeatedly applying ordinary least squares with each new observation collected to update the regression model.

See also the information in Section 4.2.15.

## References

- Yitong Liu, Zhengshuo Li, and Shumin Sun. “A Data-Driven Method for Online Constructing Linear Power Flow Model”. In: *IEEE Transactions on Industry Applications* (2023)
- Manoj Badoni, Alka Singh, and Bhim Singh. “Variable Forgetting Factor Recursive Least Square Control Algorithm for DSTATCOM”. in: *IEEE Transactions on Power Delivery* 30.5 (2015), pp. 2353–2361. DOI: [10.1109/TPWRD.2015.2422139](https://doi.org/10.1109/TPWRD.2015.2422139)



## 4.3 Partial Least Squares Regression Family

This class of functions covers the following algorithms:

- Ordinary partial least squares with SIMPLS (PLS\_SIM)
- Ordinary partial least squares with SIMPLS using rank of  $\mathbf{X}$  (PLS\_SIMRX)
- Ordinary partial least squares with NIPALS (PLS\_NIP)
- Partial least squares bundling known/unknown variables and replacing slack bus's power injection (PLS\_BDL)
- Partial least squares bundling known/unknown variables (PLS\_BDLY2)
- Partial least squares bundling known and unknown variables, based on the open-source code of [20] (PLS\_BDLOpen)
- Recursive partial least squares with NIPALS (PLS\_REC)
- Recursive partial least squares with NIPALS, using forgetting factors for observations (PLS\_RECW)
- Repeated partial least squares with NIPALS (PLS\_REP)
- Partial least squares with clustering (PLS\_CLS)

### More About

The ordinary partial least squares approach projects  $\mathbf{X}$  and  $\mathbf{Y}$  onto lower-dimensional spaces defined by their orthogonal score vectors, thereby removing the correlated components within the original datasets. The projection amounts to decomposing  $\mathbf{X}$  and  $\mathbf{Y}$  into

$$\begin{aligned}\mathbf{X} &= \mathbf{T}\mathbf{C}^\top + \mathbf{E} \\ \mathbf{Y} &= \mathbf{U}\mathbf{R}^\top + \mathbf{F}\end{aligned}$$

where  $\mathbf{T} \in \mathbb{R}^{N_s \times N_p}$  and  $\mathbf{U} \in \mathbb{R}^{N_s \times N_p}$  consist of  $N_p$  score components extracted from  $\mathbf{X}$  and  $\mathbf{Y}$ ,  $\mathbf{C} \in \mathbb{R}^{N_x \times N_p}$  and  $\mathbf{R} \in \mathbb{R}^{N_y \times N_p}$  denote the loading matrices of  $\mathbf{X}$  and  $\mathbf{Y}$ .  $\mathbf{E} \in \mathbb{R}^{N_s \times N_x}$  and  $\mathbf{F} \in \mathbb{R}^{N_s \times N_y}$  are the residuals for  $\mathbf{X}$  and  $\mathbf{Y}$ .

The solution of the ordinary partial least squares regression is explicitly given by

$$\hat{\beta} = \mathbf{X}^\top \mathbf{U} \left( \mathbf{T}^\top \mathbf{X} \mathbf{X}^\top \mathbf{U} \right)^{-1} \mathbf{T}^\top \mathbf{Y}$$

Notes that when  $N_p = N_x$ , i.e., the number of score components equals the number of variables in  $\mathbf{X}$ , the ordinary partial least squares regression degrades to ordinary least squares regression Qin [47].

### 4.3.1 Ordinary Partial Least Squares with SIMPLS (PLS\_SIM)

#### Tips

- Unless  $N_y = 1$ , SIMPLS and NIPALS generate very slight differences in  $\beta$ . Results from Alin [48] show that of the two, SIMPLS enjoys higher computational efficiency.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_SIM');
```

```
>> model = daline.fit(data, 'method.name', 'PLS_SIM', 'variable.predictor', {'P'}, 'variable.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'PLS_SIM', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_SIM(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_PLS_SIM(data, opt);
```

## More About

The SIMPLS algorithm, so named because it is a “straightforward implementation of a statistically inspired modification of the PLS method,” is one approach to performing the PLS decompositions. It does so by calculating the score components  $\mathbf{T}$  and  $\mathbf{U}$  directly as linear combinations of the original variables in  $\mathbf{X}$  and  $\mathbf{Y}$ .

Note that PLS\_SIM centers  $\mathbf{X}$  first before taking its rank as the number of score components, instead of directly taking the rank of  $\mathbf{X}$ .

## References

- Yi Tan et al. “Linearizing power flow model: A hybrid physical model-driven and data-driven approach”. In: *IEEE Transactions on Power Systems* 35.3 (2020), pp. 2475–2478
- Aylin Alin. “Comparison of PLS algorithms when number of objects is much larger than number of variables”. In: *Statistical papers* 50.4 (2009), pp. 711–720
- Sijmen De Jong. “SIMPLS: an alternative approach to partial least squares regression”. In: *Chemometrics and intelligent laboratory systems* 18.3 (1993), pp. 251–263

## 4.3.2 Ordinary Partial Least Squares with SIMPLS Using Rank of $\mathbf{X}$ (PLS\_SIMRX)

### Tips

- This function should be used for comparison with PLS\_SIM in terms of accuracy. Preliminary empirical testing suggests that PLS\_SIM generally outperforms PLS\_SIMRX significantly. One major reason is that PLS\_SIM centers  $\mathbf{X}$  first before taking its rank as the number of score components, while PLS\_SIMRX directly takes the rank of  $\mathbf{X}$  as the number of score components.

## Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_SIMRX');
```

```
>> model = daline.fit(data, 'method.name', 'PLS_SIMRX', 'variable.predictor', {'P'}, '
    variable.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'PLS_SIMRX', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_SIMRX(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_PLS_SIMRX(data, opt);
```

## More About

This method sets the number of score components equal to the rank of  $\mathbf{X}$ , which is typical for generic applications of PLS. In contrast, PLS\_SIM uses the rank of the *centered*  $\mathbf{X}$  instead. See also the information in Section 4.3.1.

### 4.3.3 Ordinary Partial Least Squares with NIPALS (PLS\_NIP)

#### Additional inputs

Table 4.17: Table of parameters specific to ordinary partial least squares with NIPALS.

Parameter	Format	Default	Description
PLS.outerTol	float	1e-12	The tolerance for stopping the outer iteration loop of NIPALS; this value should be very small.
PLS.innerTol	float	1e-12	The tolerance for stopping the inner iteration loop of NIPALS; this value should be very small.

## Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_NIP');
```

```
>> model = daline.fit(data, 'method.name', 'PLS_NIP', 'variable.predictor', {'P'}, '
    variable.response', {'PF', 'Vm'}, 'PLS.innerTol', 1e-10);
```

```
>> opt = daline.setopt('method.name', 'PLS_NIP', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'}, 'PLS.innerTol', 1e-10);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_NIP(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'QF'}, 'PLS.innerTol', 1e-10);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'}, '
    PLS.innerTol', 1e-10);
>> model = func_algorithm_PLS_NIP(data, opt);
```

## More About

The nonlinear iterative partial least squares (NIPALS) algorithm is the classical method used in PLS, and works by constructing deflated data matrices of  $\mathbf{X}$  and  $\mathbf{Y}$ ; i.e., each iteration of NIPALS yields new data sets  $\mathbf{X}_t$  and  $\mathbf{Y}_t$  representing matrices of residuals after regressing all variables on the current set of score components  $\mathbf{T}$ .

In the outer iteration loop of NIPALS,  $\mathbf{T}$  and  $\mathbf{U}$  are updated until  $\mathbf{T}$  converges within the tolerance given by `PLS.outerTol`.  $\hat{\beta}$  is computed in the inner iteration loop, which is exited from when the deflated  $\mathbf{X}$  (i.e., the residuals) is smaller than the specified `PLS.innerTol`.

## References

- S Joe Qin. “Partial least squares regression for recursive system identification”. In: *Proceedings of 32nd IEEE Conference on Decision and Control*. IEEE. 1993, pp. 2617–2622
- Herman Wold. “Path models with latent variables: The NIPALS approach”. In: *Quantitative sociology*. Elsevier, 1975, pp. 307–357

### 4.3.4 Partial Least Squares Bundling Known/Unknown Variables and Replacing Slack Bus’s Power Injection (PLS\_BDL)

#### Tips

- This method fixes the predictors as  $V_m$ ,  $P$ ,  $Q$  and the responses as  $V_m$ ,  $V_a$ ,  $P$ ,  $Q$ ,  $PF$ ,  $PT$ ,  $QF$ ,  $QT$  in `opt.variable.predictor` and `opt.variable.response` respectively. Changing the inputs to these function arguments will have no effect on the final output of `PLS_BDL`.
- The idea behind “bundling” is to group known (i.e., independent) and unknown (i.e., dependent) variables in a way that makes DPFL models resilient to bus-type variations. For example, this occurs when a PQ bus changes to a PV bus, such that its reactive power injections become unknown but its voltages become known. With bundling, the  $\beta$  learned under previous bus-type paradigms is still valid.

- Large linearization errors or even failures may occur if the  $\hat{\beta}_{22}$  described below is nearly singular. This may arise in scenarios where, for instance, the voltages at certain PV buses are fixed in the training dataset.

### Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_BDL');
```

```
>> opt = daline.setopt('method.name', 'PLS_BDL');
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_BDL(data);
```

### More About

In this approach,  $\mathbf{X}$  is separated into  $\mathbf{X}_1 \in \mathbb{R}^{N_s \times N_{x_1}}$  and  $\mathbf{X}_2 \in \mathbb{R}^{N_s \times N_{x_2}}$ , where  $N_x = N_{x_1} + N_{x_2}$ .  $\mathbf{X}_1$  collects the measurements of the active/reactive power injections at PQ buses and the active power injections at PV buses, while  $\mathbf{X}_2$  collects the observations of the voltages of the slack and PV buses.

Similarly,  $\mathbf{Y}$  splits into  $\mathbf{Y}_1 \in \mathbb{R}^{N_s \times N_{y_1}}$  and  $\mathbf{Y}_2 \in \mathbb{R}^{N_s \times N_{y_2}}$ , where  $N_y = N_{y_1} + N_{y_2}$ .  $\mathbf{Y}_1$  contains the realizations of the angles of PQ and PV buses, the voltages of PQ buses, and the active power injection of the slack bus.  $\mathbf{Y}_2$  consists of the measurements for the reactive power injections at the slack and PV buses.

The following relationship is then estimated via partial least squares, using the SIMPLS algorithm:

$$\begin{bmatrix} \mathbf{Y}_1 & \mathbf{X}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{X}_1 & \mathbf{Y}_2 \end{bmatrix} \begin{bmatrix} \beta_{11} & \beta_{12} \\ \beta_{21} & \beta_{22} \end{bmatrix}$$

Subsequently, for any new input  $\mathbf{x} = [\mathbf{x}_1^\top \ \mathbf{x}_2^\top]^\top$ , the estimated value of  $\mathbf{y} = [\mathbf{y}_1^\top \ \mathbf{y}_2^\top]^\top$  is obtained through

$$\begin{aligned} \hat{\mathbf{y}}_2^\top &= (\mathbf{x}_2^\top - \mathbf{x}_1^\top \hat{\beta}_{12}) \hat{\beta}_{22}^{-1} \\ \hat{\mathbf{y}}_1^\top &= \mathbf{x}_1^\top \hat{\beta}_{11} + \hat{\mathbf{y}}_2^\top \hat{\beta}_{21} \end{aligned}$$

Note that  $\hat{\beta}_{22}$  is a square matrix, since  $N_{x_2} = N_{y_2}$ .

### References

Yuxiao Liu et al. “Data-driven power flow linearization: A regression approach”. In: *IEEE Transactions on Smart Grid* 10.3 (2018), pp. 2569–2580

### 4.3.5 Partial Least Squares Bundling Known/Unknown Variables (PLS\_SIMY2)

#### Tips

- Here, the active power of the slack bus (i.e.,  $P_{\text{ref}}$ ) is grouped under  $\mathbf{Y}_2$ , while in PLS\_BDL,  $P_{\text{ref}}$  sits in  $\mathbf{Y}_1$  as a response. Thus, this function should be used for comparison with PLS\_BDL to examine the significance of moving  $P_{\text{ref}}$  from being a predictor to being a response variable.
- Large linearization errors or even failures may occur if the  $\hat{\beta}_{22}$  described in Section 4.3.4 is nearly singular. This may arise in scenarios where, for instance, the voltages at certain PV buses are fixed in the training dataset.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_BDLY2');
```

```
>> opt = daline.setopt('method.name', 'PLS_BDLY2');
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_BDLY2(data);
```

#### More About

In this version of bundling, to ensure that  $\hat{\beta}_{22}$  remains as a square matrix even after the inclusion of the active power of the slack bus in  $\mathbf{Y}_2$ , the voltage angle of the slack bus is added to  $\mathbf{X}_2$ . See also the information in Section 4.3.4.

### 4.3.6 Partial Least Squares Bundling Known/Unknown Variables: the Open-source Version (PLS\_BDLopen)

#### Tips

- This method fixes the predictors as  $\mathbf{P}$ ,  $\mathbf{Q}$  and the responses as  $\mathbf{V}_m$ ,  $\mathbf{V}_a$  in `opt.variable.predictor` and `opt.variable.response` respectively. Changing the inputs to these function arguments will have no effect on the final output of PLS\_BDLopen.
- It should be noted that PLS\_BDLopen exclusively produces errors in  $\mathbf{V}_m$  and  $\mathbf{V}_a$ . This limitation stems from the limited capability of the open-source code associated with [20].
- PLS\_BDLopen wraps code from the open source MATLAB scripts provided by Liu et al. [20], without modification. It deviates from PLS\_BDL in that:
  - (i) Only active and reactive power injections (i.e.,  $\mathbf{P}$ ,  $\mathbf{Q}$ ) are used as predictor variables.
  - (ii) The active power injection of the slack bus is assumed to be zero.
  - (iii) The component number, i.e.,  $N_p$ , used in PLS\_BDLopen is neither the rank of  $\mathbf{X}$ , nor the rank of the centered  $\mathbf{X}$ . PLS\_BDL uses rank of the centered  $\mathbf{X}$  as  $N_p$ .
- Preliminary empirical testing suggests that PLS\_BDL generally achieves significantly higher linearization accuracy compared to PLS\_BDLopen.

- Large linearization errors or even failures may occur if the  $\hat{\beta}_{22}$  described in Section 4.3.4 is nearly singular. This may arise in scenarios where, for instance, the voltages at certain PV buses are fixed in the training dataset.

## Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_BDLopen');
```

```
>> opt = daline.setopt('method.name', 'PLS_BDLopen');
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_BDLopen(data);
```

## More About

For technical details explaining the concept of bundling, see also the information in Section 4.3.4.

## References

Yuxiao Liu et al. “Data-driven power flow linearization: A regression approach”. In: *IEEE Transactions on Smart Grid* 10.3 (2018), pp. 2569–2580

## 4.3.7 Recursive Partial Least Squares with NIPALS (PLS\_REC)

### Additional inputs

Table 4.18: Table of parameters specific to recursive partial least squares with NIPALS.

Parameter	Format	Default	Description
PLS.recursivePercentage	float	20	Percentage of the training data set that comprises new data. Each new observation will be learned recursively.
PLS.outerTol	float	1e-9	The tolerance for stopping the outer iteration loop of NIPALS; this value should be very small.
PLS.innerTol	float	1e-9	The tolerance for stopping the inner iteration loop of NIPALS; this value should be very small.

## Tips

- The decompositions of the updated results have a fixed computational cost because  $\widetilde{\mathbf{X}}[t+1]$  and  $\widetilde{\mathbf{Y}}[t+1]$  have  $N_p + 1$  rows for all  $t$ . In contrast, the numbers of rows of  $\mathbf{X}[t+1]$  and  $\mathbf{Y}[t+1]$  are both  $t+1$ , grow linearly with  $t$ , and can reach the total number of observations  $N_s$ . Given that

$N_p \ll N_s$  holds outside high-dimensional settings, `func_algorithm_PLS_REC` should be significantly less computationally intensive than `func_algorithm_PLS_REP`.

### Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_REC');
```

```
>> model = daline.fit(data, 'method.name', 'PLS_REC', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'PLS.recursivePercentage', 50, 'PLS.innerTol', 1e-10);
```

```
>> opt = daline.setopt('method.name', 'PLS_REC', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'PLS.recursivePercentage', 50, 'PLS.innerTol', 1e-10);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_REC(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'PLS.recursivePercentage', 50, 'PLS.innerTol', 1e-10);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'},
    'PLS.recursivePercentage', 50, 'PLS.innerTol', 1e-10);
>> model = func_algorithm_PLS_REC(data, opt);
```

### More About

Suppose that  $\mathbf{X}[t]$  and  $\mathbf{Y}[t]$ , as determined by `PLS.recursivePercentage`, have already been decomposed into

$$\begin{aligned}\mathbf{X}[t] &= \mathbf{T}[t]\mathbf{C}[t]^\top + \mathbf{E}[t] \\ \mathbf{Y}[t] &= \mathbf{U}[t]\mathbf{R}[t]^\top + \mathbf{F}[t]\end{aligned}$$

with

$$\begin{aligned}\mathbf{T}[t] &= [\mathbf{t}[t]_1 \cdots \mathbf{t}[t]_{N_p}] \\ \mathbf{U}[t] &= [\mathbf{u}[t]_1 \cdots \mathbf{u}[t]_{N_p}]\end{aligned}$$

When the new measurements at time step  $t + 1$ , i.e.,  $\mathbf{x}_{t+1}$  and  $\mathbf{y}_{t+1}$ , are available, this method avoids having to re-decompose  $\mathbf{X}[t + 1]$  and  $\mathbf{Y}[t + 1]$ . Instead, it computes

$$\widetilde{\mathbf{X}}[t + 1] = \begin{bmatrix} \mathbf{C}[t]^\top \\ \mathbf{x}_{t+1} \end{bmatrix}, \quad \widetilde{\mathbf{Y}}[t + 1] = \begin{bmatrix} \mathbf{I}[t]\mathbf{R}[t]^\top \\ \mathbf{y}_{t+1} \end{bmatrix}$$



where

$$\mathbf{\Gamma}[t] = \text{diag}(\gamma[t]_1 \cdots \gamma[t]_{N_p})$$

with

$$\gamma[t]_i = \frac{\mathbf{u}[t]_i^\top \mathbf{t}[t]_i}{\mathbf{t}[t]_i^\top \mathbf{t}[t]_i}$$

The decomposition results are denoted by

$$\begin{aligned}\widetilde{\mathbf{X}}[t+1] &= \widetilde{\mathbf{T}}[t+1]\widetilde{\mathbf{C}}[t+1]^\top + \widetilde{\mathbf{E}}[t+1] \\ \widetilde{\mathbf{Y}}[t+1] &= \widetilde{\mathbf{U}}[t+1]\widetilde{\mathbf{R}}[t+1]^\top + \widetilde{\mathbf{F}}[t+1]\end{aligned}$$

Substituting  $\widetilde{\mathbf{X}}[t+1]$  and  $\widetilde{\mathbf{Y}}[t+1]$  as well as  $\widetilde{\mathbf{T}}[t+1]$  and  $\widetilde{\mathbf{U}}[t+1]$  into the usual explicit solution for ordinary partial least squares,  $\hat{\boldsymbol{\beta}} = \mathbf{X}^\top \mathbf{U} (\mathbf{T}^\top \mathbf{X} \mathbf{X}^\top \mathbf{U})^{-1} \mathbf{T}^\top \mathbf{Y}$ , generates the revised regression coefficients.

The NIPALS algorithm is used to realize all decompositions in this method. For more details on NIPALS, see also the information in Section 4.3.3.

## References

- Severin Nowak, Yu Christine Chen, and Liwei Wang. “Measurement-based optimal DER dispatch with a recursively estimated sensitivity model”. In: *IEEE Transactions on Power Systems* 35.6 (2020), pp. 4792–4802
- S Joe Qin. “Recursive PLS algorithms for adaptive data modeling”. In: *Computers & Chemical Engineering* 22.4-5 (1998), pp. 503–514

### 4.3.8 Recursive Partial Least Squares with NIPALS with Forgetting Factors (PLS\_RECW)

#### Additional inputs

Table 4.19: Table of parameters specific to recursive partial least squares with NIPALS, using forgetting factors for observations.

Parameter	Format	Default	Description
PLS.omega	float	0.6	Forgetting factor value $\varpi$ with a valid range from 0 to 1. The default value of 0.6 is from Nowak, Chen, and Wang [21].
PLS.recursivePercentage	float	20	Percentage of the training data set that comprises new data. Each new observation will be learnt recursively.
PLS.outerTol	float	1e-9	The tolerance for stopping the outer iteration loop of NIPALS; this value should be very small.
PLS.innerTol	float	1e-9	The tolerance for stopping the inner iteration loop of NIPALS; this value should be very small.

## Tips

- If `PLS.omega` is set to a value less than 1, the weights of earlier measurements decrease exponentially with the number of updates. Although this helps in situations where the system operating point is changing, over time, it also causes earlier observations to take on practically insignificant weights. Thus, to optimize prediction accuracy, the use of cross-validation for the forgetting factor `PLS.omega` is recommended.

## Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_RECW');
```

```
>> model = daline.fit(data, 'method.name', 'PLS_RECW', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'PLS.omega', 0.95, 'PLS.recursivePercentage',
    30);
```

```
>> opt = daline.setopt('method.name', 'PLS_RECW', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'PLS.omega', 0.95, 'PLS.recursivePercentage', 30)
;
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_RECW(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'PLS.omega', 0.95, 'PLS.recursivePercentage', 30);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'PLS.omega', 0.95, 'PLS.recursivePercentage', 30);
>> model = func_algorithm_PLS_RECW(data, opt);
```

## More About

A forgetting factor,  $\varpi$ , is used to update the growing datasets  $\tilde{\mathbf{X}}[t+1]$  and  $\tilde{\mathbf{Y}}[t+1]$  as

$$\tilde{\mathbf{X}}[t+1] = \begin{bmatrix} \varpi \mathbf{C}[t]^\top \\ \mathbf{x}_{t+1} \end{bmatrix}, \tilde{\mathbf{Y}}[t+1] = \begin{bmatrix} \varpi \mathbf{\Gamma}[t] \mathbf{R}[t]^\top \\ \mathbf{y}_{t+1} \end{bmatrix}$$

Parameter  $\varpi$  belongs to the range  $(0, 1]$  and can be set via cross-validation. Values of  $\varpi$  smaller than 1 cause contributions from earlier measurements to factor in less during incremental updating of the partial least squares model.

## References

Severin Nowak, Yu Christine Chen, and Liwei Wang. “Measurement-based optimal DER dispatch with a recursively estimated sensitivity model”. In: *IEEE Transactions on Power Systems* 35.6 (2020), pp. 4792–4802

### 4.3.9 Repeated Partial Least Squares with NIPALS (PLS\_REP)

#### Additional inputs

Table 4.20: Table of parameters specific to repeated partial least squares with NIPALS.

Parameter	Format	Default	Description
PLS.recursivePercentage	float	20	Percentage of the training data set that comprises new data. Each new observation will be learnt recursively.
PLS.outerTol	float	1e-9	The tolerance for stopping the outer iteration loop of NIPALS; this value should be very small.
PLS.innerTol	float	1e-9	The tolerance for stopping the inner iteration loop of NIPALS; this value should be very small.

#### Tips

- This function should be used for comparison with PLS\_REC in terms of computational time and accuracy.
- In this method, the decomposition of the updated results has a computational cost that scales with the size of the training data set.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_REP');
```

```
>> model = daline.fit(data, 'method.name', 'PLS_REP', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'PLS.recursivePercentage', 50, 'PLS.innerTol', 1
    e-10);
```

```
>> opt = daline.setopt('method.name', 'PLS_REP', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'PLS.recursivePercentage', 50, 'PLS.innerTol', 1e
    -10);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_REP(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'PLS.recursivePercentage', 50, 'PLS.innerTol', 1e-10);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'PLS.recursivePercentage', 50, 'PLS.innerTol', 1e-10);
>> model = func_algorithm_PLS_REP(data, opt);
```

### More About

Suppose that  $\mathbf{X}[t]$  and  $\mathbf{Y}[t]$ , as determined by `PLS.recursivePercentage`, have already been decomposed into

$$\mathbf{X}[t] = \mathbf{T}[t]\mathbf{C}[t]^\top + \mathbf{E}[t]$$

$$\mathbf{Y}[t] = \mathbf{U}[t]\mathbf{R}[t]^\top + \mathbf{F}[t]$$

With each new observation, the decomposition is re-performed using an updated:

$$\mathbf{X}[t+1] = \begin{bmatrix} \mathbf{X}[t] \\ \mathbf{x}_{t+1} \end{bmatrix}, \quad \mathbf{Y}[t+1] = \begin{bmatrix} \mathbf{Y}[t] \\ \mathbf{y}_{t+1} \end{bmatrix}$$

Substituting  $\mathbf{X}[t+1]$  and  $\mathbf{Y}[t+1]$  as well as  $\mathbf{T}[t+1]$  and  $\mathbf{U}[t+1]$  into the usual explicit solution for ordinary partial least squares,  $\hat{\boldsymbol{\beta}} = \mathbf{X}^\top \mathbf{U} (\mathbf{T}^\top \mathbf{X} \mathbf{X}^\top \mathbf{U})^{-1} \mathbf{T}^\top \mathbf{Y}$ , generates the revised regression coefficients.

### 4.3.10 Partial Least Squares with Clustering (PLS\_CLS)

#### Additional inputs

Table 4.21: Table of parameters specific to partial least squares with clustering.

Parameter	Format	Default	Description
PLS.parallel	binary	1	1: Use parallel computation; otherwise 0.
PLS.clusNumInterval	vector/integer	[2:1:10]	The discrete range of the number of clusters to be tuned by cross-validation. If a scalar integer is given, then use it directly without tuning.
PLS.cvNumFold	integer	10	The number of folds for cross-validation tuning of the number of clusters to be used in K-means. This number must be divisible by the number of training samples.
PLS.fixKmeans	binary	1	1: Fix the random seed for K-means in PLS_CLS for consistency in clustering; otherwise 0.
PLS.fixCV	binary	1	1: Fix the random seed for partitioning data in CV; otherwise 0.
PLS.fixSeed	integer	88	Random seed number for K-means and CV partitioning.

#### Tips

- The primary application of this method arises when separate linear power flow models can be fit from different power system operating modes. Due to frequent ambiguity in the cross-validation-determined optimal number of clusters to use in regression, this method works best when the number of operating modes, and thus the number of clusters to be used, is already known.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'PLS_CLS');
```

```
>> model = daline.fit(data, 'method.name', 'PLS_CLS', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'PLS.fixKmeans', 0, 'PLS.clusNumInterval',
    [6:2:12], 'PLS.cvNumFold', 5);
```

```
>> opt = daline.setopt('method.name', 'PLS_CLS', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'PLS.fixKmeans', 0, 'PLS.clusNumInterval',
    [6:2:12], 'PLS.cvNumFold', 5);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PLS_CLS(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'PLS.fixKmeans', 0, 'PLS.clusNumInterval', [6:2:12], 'PLS.
    cvNumFold', 5);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'PLS.fixKmeans', 0, 'PLS.clusNumInterval', [6:2:12], 'PLS.cvNumFold', 5);
>> model = func_algorithm_PLS_CLS(data, opt);
```

### More About

$\mathbf{X}$  is first divided into  $K$  clusters using K-means clustering. The ordinary partial least squares method using SIMPLS is subsequently implemented on the partitioned data from each of the  $K$  clusters, such that the regression coefficients of the  $k$ -th segment are found as

$$\hat{\beta}(k) = \mathbf{X}(k)^\top \mathbf{U}(k) \left( \mathbf{T}(k)^\top \mathbf{X}(k) \mathbf{X}(k)^\top \mathbf{U}(k) \right)^{-1} \mathbf{T}(k)^\top \mathbf{Y}(k), \forall k$$

with  $\mathbf{U}$  and  $\mathbf{T}$  calculated as normal for each partition of the data.

To apply the resulting piecewise model, identify the cluster that a given input  $\mathbf{x}$  belongs to via

$$k = \arg \min_j \|\mathbf{x} - \boldsymbol{\mu}(j)\|_2^2$$

where  $\boldsymbol{\mu}(j)$  is the centroid of cluster  $j$ . Then, use the corresponding  $\hat{\beta}$  to generate the prediction of  $\mathbf{y}$  as  $\hat{\mathbf{y}} = \hat{\beta}(k)^\top \mathbf{x}$ .

## 4.4 Ridge Regression Family

This class of functions covers the following algorithms:

- Ordinary ridge regression (**RR**)
- Ordinary ridge regression with voltage-angle coupling (**RR\_VCS**)
- Ordinary ridge regression with K-plane clustering (**RR\_KPC**)
- Locally weighted ridge regression (**RR\_WEI**)

They share common parameters listed in Table 4.22.

**NOTE** The ridge regression model does not include a constant term. In DALINE, all functions in the ridge regression class besides **RR\_KPC** automatically remove the column of 1s in the predictor data.

Table 4.22: Table of parameters common to all ridge-regression-based algorithms.

Parameter	Format	Default	Description
<b>RR.lambdaInterval</b>	vector/float	1e-10	The discrete range of tuning the regularization factor using cross-validation, e.g., [0:1e-3:0.02]. If a scalar float is given, then use it directly without tuning.
<b>RR.cvNumFold</b>	integer	10	The number of folds for hyperparameter tuning via cross-validation, e.g., for the regularization factor. This number must be divisible by the number of training samples.
<b>RR.fixCV</b>	binary	1	1: Fix the random seed for partitioning data in CV; otherwise 0.
<b>RR.fixSeed</b>	integer	88	Random seed number for K-means and CV partitioning.

### Tips

- Using a larger number of folds in **RR.cvNumFold** leads to cross-validation errors that are closer estimates of the generalization error. However, doing so also increases the computational burden of CV linearly. Typically, as a compromise, the number of folds is set to 5 or 10.
- The regularization factor usually takes on a very small value. Larger values introduce bias and result in linearization coefficient shrinkage. However, they also reduce the variance of ridge regression estimates. Consequently, using cross-validation for this parameter, via giving a range for **RR.lambdaInterval**, is recommended.
  - In the event that more than one regularization factor returns the smallest CV error, the CV subroutine picks the factor that appears first in **RR.lambdaInterval**.

### 4.4.1 Ordinary Ridge Regression (RR)

#### Examples

```
>> model = daline.fit(data, 'method.name', 'RR');
```

```
>> model = daline.fit(data, 'method.name', 'RR', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-3:0.1]);
```

```
>> opt = daline.setopt('method.name', 'RR', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-3:0.1]);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_RR(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-3:0.1]);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-3:0.1]);
>> model = func_algorithm_RR(data, opt);
```

## More About

This method introduces a regularization penalty (a.k.a., the Tikhonov-Phillips regularization) to the objective function of the regression model. The resulting objective becomes

$$\min_{\beta} \|\mathbf{Y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_2^2$$

where  $\lambda \in \mathbb{R}$  is a preset tuning parameter, i.e., the regularization factor. Correspondingly, the solution for  $\beta$  is

$$\hat{\beta} = \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}\right)^{-1} \mathbf{X}^\top \mathbf{Y}$$

where  $\mathbf{I}$  is an identity matrix of appropriate dimension. The parameter  $\lambda$  tunes the diagonal elements (i.e., the ridge) in  $\mathbf{X}^\top \mathbf{X}$  to ensure the invertibility of this matrix.

## References

Yanbo Chen, Chao Wu, and Junjian Qi. “Data-Driven Power Flow Method Based on Exact Linear Regression Equations”. In: *Journal of Modern Power Systems and Clean Energy* (2021)



## 4.4.2 Ordinary Ridge Regression with the Voltage-angle Coupling (RR\_VCS)

### Additional inputs

Table 4.23: Table of parameters specific to ordinary ridge regression with VCS coordinate transformation.

Parameter	Format	Default	Description
<code>opt.idx</code>	function/struct	<code>func_define_idx</code>	A struct (or function calling a struct) that contains the column indices of the from- and to-buses in a MATPOWER-like <code>mpc.branch</code> matrix. By default, it uses the built-in index in MATPOWER.

### Tips

- This method fixes the predictors as  $V_m2$ ,  $P$ ,  $Q$  and the responses as  $V_m2$ ,  $V_i V_j \sin(\theta_{ij})$ , and  $V_i V_j \sin(\theta_{ij})$  in `opt.variable.predictor` and `opt.variable.response` respectively. Changing the inputs to these function arguments will have no effect on the output. From the model responses, the possible outputs are then the voltages of all PQ buses, PF, QF, PT, and QT.

### Examples

```
>> model = daline.fit(data, 'method.name', 'RR_VCS');
```

```
>> model = daline.fit(data, 'method.name', 'RR_VCS', 'RR.lambdaInterval', [0:5e-3:0.1]);
```

```
>> opt = daline.setopt('method.name', 'RR_VCS', 'RR.lambdaInterval', [0:5e-3:0.1]);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_RR_VCS(data, 'RR.lambdaInterval', [0:5e-3:0.1]);
```

```
>> opt = daline.setopt('RR.lambdaInterval', [0:5e-3:0.1]);
>> model = func_algorithm_RR_VCS(data, opt);
```

### More About

This method implements coordinate transformation in the form of voltage-angle coupling and voltage squaring to transform the non-linear AC power flow equations into

$$P_i = G_{ii}U_i + \sum_{j \in k(i), j \neq i} (G_{ij}R_{ij} + B_{ij}C_{ij})$$

and

$$Q_i = -B_{ii}U_i + \sum_{j \in k(i), j \neq i} (G_{ij}C_{ij} - B_{ij}R_{ij})$$

where  $R_{ij} = V_i V_j \cos(\theta_{ij})$ ,  $C_{ij} = V_i V_j \sin(\theta_{ij})$ , and  $U_i = V_i^2$ . These linearizations can be used to represent PQ buses. On the other hand, PV buses are represented by the first equation above and

$$V_{i,PV}^2 = U_i$$

Subsequently, voltages for the PQ buses and line flows can be computed from the predicted values of  $R_{ij}$ ,  $C_{ij}$ , and  $U_i$ .

## References

Yanbo Chen, Chao Wu, and Junjian Qi. “Data-Driven Power Flow Method Based on Exact Linear Regression Equations”. In: *Journal of Modern Power Systems and Clean Energy* (2021)

### 4.4.3 Ordinary Ridge Regression with K-plane Clustering (RR\_KPC)

#### Additional inputs

Table 4.24: Table of parameters specific to ridge regression with K-plane clustering.

Parameter	Format	Default	Description
RR.etaInterval	vector/float	logspace(2, 5, 4)	The discrete range of tuning $\eta$ (as explained below) using cross-validation, e.g., logspace(2, 5, 4). If a scalar float is given, then use it directly without tuning.
RR.kplaneMaxIter	integer	1e5	The maximum number of iterations allowed when performing K-plane clustering. This method usually converges within 10 iterations.
RR.fixKmeans	binary	1	1: Fix the random seed for K-means in RR_KPC for consistency in clustering; otherwise 0 (K-means is used for initialization).
RR.fixSeed	integer	88	Random seed number for K-means.
RR.clusNumInterval	vector/integer	[2:1:10]	The discrete range of the number of clusters for tuning via cross-validation. If an integer is given, the cluster number is fixed to this value.

## Tips

- Tuning the regularization factor  $\lambda$ ,  $\eta$ , and the number of clusters by cross-validation can be very time-consuming. Instead of setting the regularization factor or  $\eta$  to an arbitrary value, it is recommended that they be tuned in tandem, as their optimal values are codependent.

- Assuming a sufficiently large training data set, this method generally returns lower errors as the number of clusters increases. However, gains in accuracy may taper off after a point.

### Examples

```
>> model = daline.fit(data, 'method.name', 'RR_KPC');
```

```
>> model = daline.fit(data, 'method.name', 'RR_KPC', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-2:0.1], 'RR.etaInterval', logspace(-3, 3, 7), 'RR.clusNumInterval', [2:2:10]);
```

```
>> opt = daline.setopt('method.name', 'RR_KPC', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-2:0.1], 'RR.etaInterval', logspace(-3, 3, 7), 'RR.clusNumInterval', [2:2:10]);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_RR_KPC(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-2:0.1], 'RR.etaInterval', logspace(-3, 3, 7), 'RR.clusNumInterval', [2:2:10]);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-2:0.1], 'RR.etaInterval', logspace(-3, 3, 7), 'RR.clusNumInterval', [2:2:10]);
>> model = func_algorithm_RR_KPC(data, opt);
```

### More About

This method fits the AC power flow model in a piecewise manner, with each piece represented by a ridge regression model trained on its own cluster of data points. Unlike clustering-based least squares in 4.2.11, here, ridge regression is embedded into each iteration of the clustering.

For  $[\mathbf{S}(k)^n, \boldsymbol{\mu}(k)^n, \hat{\boldsymbol{\beta}}(k)^n]$ , where  $n$  indicates the iteration number,  $\mathbf{S}(k)^n$  consists of  $\mathbf{x}_i$  belonging to cluster  $k$ ,  $\boldsymbol{\mu}(k)^n$  denotes the centroid of cluster  $k$ , and  $\hat{\boldsymbol{\beta}}(k)^n$  represents the regression coefficient for cluster  $k$ , in each iteration,

- (i)  $\hat{\boldsymbol{\beta}}(k)^n$  is updated via

$$\hat{\boldsymbol{\beta}}(k)^{n+1} = [\mathbf{X}(k)^\top \mathbf{X}(k) + \lambda \mathbf{I}]^{-1} \mathbf{X}(k)^\top \mathbf{Y}(k), \forall k$$

where  $\mathbf{X}(k)$  is composed of  $\mathbf{x}_i \in \mathbf{S}(k)^n$  while  $\mathbf{Y}(k)$  collects  $\mathbf{y}_i$  that correspond to  $\mathbf{x}_i \in \mathbf{S}(k)^n$ .

- (ii)  $\boldsymbol{\mu}(k)^n$  is updated via

$$\boldsymbol{\mu}(k)^{n+1} = \sum_{\mathbf{x}_i \in \mathbf{S}(k)^n} \frac{\mathbf{x}_i}{|\mathbf{S}(k)^n|}, \forall k$$

where  $|\cdot|$  denotes the cardinality function.

(iii)  $\mathbf{x}_i$  ( $\forall i$ ) is re-allocated to cluster  $k$ , where

$$k = \arg \min_j \|\mathbf{y}_i^\top - \mathbf{x}_i^\top \hat{\boldsymbol{\beta}}(j)^{n+1}\|_F^2 + \eta \|\mathbf{x}_i - \boldsymbol{\mu}(j)^{n+1}\|_F^2$$

(iv) Repeat the above steps until convergence, i.e., the variations within  $\boldsymbol{\mu}$  and  $\hat{\boldsymbol{\beta}}$  are negligible.

To apply the obtained piecewise linear power flow model, the cluster to which a given input  $\mathbf{x}$  belongs to is identified with

$$k = \arg \min_j \|\mathbf{x} - \boldsymbol{\mu}(j)\|_2^2$$

Then,  $\mathbf{x}$  is substituted into  $\hat{\mathbf{y}} = \hat{\boldsymbol{\beta}}(k)^\top \mathbf{x}$  to predict the value of  $\mathbf{y}$ .

## References

Jiaqi Chen, Wenchuan Wu, and Line A Roald. “Data-driven Piecewise Linearization for Distribution Three-phase Stochastic Power Flow”. In: *IEEE Transactions on Smart Grid* (2021)

### 4.4.4 Locally Weighted Ridge Regression (RR\_WEI)

#### Additional inputs

Table 4.25: Table of parameters specific to locally weighted ridge regression.

Parameter	Format	Default	Description
RR.tauInterval	vector/float	[0.1:1e-3:0.35]	The discrete range of $\tau$ (as explained below) for cross-validation. If a scalar is given, then use it directly without tuning.

#### Tips

- Tuning both the regularization factor  $\lambda$  and  $\tau$  by cross-validation can be very time-consuming. Usually,  $\lambda$  has a greater impact on the results of this method than  $\tau$ .

#### Examples

```
>> model = daline.fit(data, 'method.name', 'RR_WEI');
```

```
>> model = daline.fit(data, 'method.name', 'RR_WEI', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-3:0.1], 'RR.tauInterval', [0.1:1e-2:0.3]);
```

```
>> opt = daline.setopt('method.name', 'RR_WEI', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-3:0.1], 'RR.
    tauInterval', [0.1:1e-2:0.3]);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_RR_WEI(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'RR.lambdaInterval', [0:5e-3:0.1], 'RR.tauInterval',
    [0.1:1e-2:0.3]);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'RR.lambdaInterval', [0:5e-3:0.1], 'RR.tauInterval', [0.1:1e-2:0.3]);
>> model = func_algorithm_RR_WEI(data, opt);
```

## More About

This method trains local models around the target operating point(s), weighting data points that are closer to the operating point of interest more heavily. Consequently, the objective function becomes

$$\min_{\beta} \|\mathbf{W}^{\frac{1}{2}}(\mathbf{Y} - \mathbf{X}\beta)\|_2^2 + \lambda\|\beta\|_2^2$$

where  $\mathbf{W}$  is a diagonal weight matrix. If the power flow model at time step  $t$  is of interest, the  $i$ -th diagonal element of  $\mathbf{W}$  is defined as

$$w_i = e^{-\frac{d_i^2}{2\tau^2}}, \text{ with } d_i = \|\mathbf{x}_i - \mathbf{x}_t\|_2$$

The solution to the minimization problem is given by

$$\hat{\beta} = \left(\mathbf{X}^\top \mathbf{W} \mathbf{X} + \lambda \mathbf{I}\right)^{-1} \mathbf{X}^\top \mathbf{W} \mathbf{Y}$$

## References

Junbo Zhang et al. “Locally weighted ridge regression for power system online sensitivity identification considering data collinearity”. In: *IEEE Transactions on Power Systems* 33.2 (2017), pp. 1624–1634

## 4.5 Support Vector Regression Family

This class of functions covers the following algorithms:

- Ordinary support vector regression: a direct solution (SVR)
- Support vector regression with polynomial kernel (SVR\_POL)
- Support vector regression with ridge regression (SVR\_RR)
- Support vector regression with chance-constrained programming (SVR\_CCP)

### General Inputs

Table 4.26: Table of parameters common to all support vector regression algorithms that **directly solve the SVR program** (SVR, SVR\_CCP, SVR\_RR).

Parameter	Format	Default	Description
SVR.epsilon	float	1e-4	The value of $\epsilon$ ; defines the margin within which errors are discounted from the SVR loss.
SVR.omega	float	10	The value of $\omega$ , which determines the trade-off between the $\ell^2$ regularization of $\beta$ and the degree to which errors greater than $\epsilon$ are accepted.
SVR.programType	character	'indivi'	'whole' puts all responses (i.e., the number of columns in $\mathbf{Y}$ ) into one optimization program to solve for all $\hat{\beta}$ at once; 'indivi' solves for $\hat{\beta}$ individually by building one optimization program for each response.
SVR.language	character	'yalmip'	Optimization toolbox to formulate the programming problem. Choose between 'cvx' or 'yalmip'.
SVR.parallel	binary	1	1: Use parallel computation; otherwise 0. Only valid when SVR.language = 'yalmip' and SVR.programType = 'indivi', because to the best of the developers' knowledge, cvx does not support parallel computing.
SVR.solver	character	'quadprog'	Solver options: 'quadprog', 'Gurobi' ('quadprog' is built in the MATLAB Optimization Toolbox; for 'Gurobi', you need to install it manually).
SVR.cvxQuiet	binary	1	1: Suppress CVX output in the command window; otherwise 0.
SVR.yalDisplay	binary	0	1: Show YALMIP display; otherwise 0.

### General Tips

- Smaller values of SVR.epsilon,  $\epsilon$ , penalize more data points. As  $\epsilon \rightarrow 0^+$ , SVR may start behaving similarly to ridge regression with regularization hyperparameter  $\lambda = \frac{1}{2}$ . Conversely, larger  $\epsilon$  values admit larger errors in the  $\epsilon$ -insensitive loss.

- Smaller values of `SVR.omega`,  $\omega$ , cause more data points to be considered as outliers that contribute to the  $\epsilon$ -insensitive cost. In contrast, large values of  $\omega$  result in a larger  $\epsilon$  tube around the regression hyperplane that accommodate outliers better, but may represent the overall data more poorly.
- The values of  $\epsilon$  and  $\omega$  interact with each other. Ideally, hyperparameter tuning via a method such as cross-validation should be performed, but is typically tricky to do. Cross-validation is not available in the current versions of the algorithms that directly solve the SVR program. It should be noted that the value of  $\epsilon$  is also significantly influenced by whether the data are normalized — normalization alters the scale of the data, which in turn affects the metric used to differentiate between inliers and outliers.

## More About

Support vector regression adopts the  $\epsilon$ -insensitive error function

$$\max \left\{ 0, \|y_{ij} - \mathbf{x}_i^\top \boldsymbol{\beta}_j\|_1 - \epsilon \right\}$$

as its objective function, such that only absolute errors greater than  $\epsilon$  are accounted for during model fitting. The  $\epsilon$ -insensitive error offers a higher tolerance for data outliers compared to the  $\ell^1$  and  $\ell^2$  norms.

Since the  $\epsilon$ -insensitive error function is not differentiable everywhere, slack variables are introduced for relaxation. The resulting regression model becomes

$$\begin{aligned} \min_{\boldsymbol{\beta}, \xi_{ij}, \xi_{ij}^*} \quad & \frac{1}{2} \|\boldsymbol{\beta}\|_2^2 + \omega \sum_{i=1}^{N_s} \sum_{j=1}^{N_y} (\xi_{ij} + \xi_{ij}^*) \\ \text{s.t.} \quad & y_{ij} - \mathbf{x}_i^\top \boldsymbol{\beta}_j \leq \epsilon + \xi_{ij}, \quad \forall i, j \\ & \mathbf{x}_i^\top \boldsymbol{\beta}_j - y_{ij} \leq \epsilon + \xi_{ij}^*, \quad \forall i, j \\ & \xi_{ij}, \xi_{ij}^* \geq 0, \quad \forall i, j \end{aligned}$$

where hyperparameters  $\omega$  and  $\epsilon$  can be determined via cross-validation.

## References

Alex J Smola and Bernhard Schölkopf. “A tutorial on support vector regression”. In: *Statistics and computing* 14.3 (2004), pp. 199–222

### 4.5.1 Ordinary Support Vector Regression: a Direct Solution (SVR)

#### Tips

- Currently, only 'indivi' is supported as a valid argument for `SVR.programType` for the current version of DALINE.
- It is highly recommended to use 'yalmip' as the language, 'quadprog' as the solver, and 'indivi' as the type of programming.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'SVR');
```

```
>> model = daline.fit(data, 'method.name', 'SVR', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'SVR.omega', 25, 'SVR.language', 'yalmip', 'SVR.programType', 'indivi', 'SVR.solver', 'quadprog');
```

```
>> opt = daline.setopt('method.name', 'SVR', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'SVR.omega', 25, 'SVR.yalDisplay', 1);  
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_SVR(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'SVR.omega', 25, 'SVR.yalDisplay', 1);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'SVR.omega', 25, 'SVR.yalDisplay', 1);  
>> model = func_algorithm_SVR(data, opt);
```

## References

Jiaqi Chen et al. “Robust Data-driven Linearization for Distribution Three-phase Power Flow”. In: *2020 IEEE 4th Conference on Energy Internet and Energy System Integration (EI2)*. IEEE. 2020, pp. 1527–1532



## 4.5.2 Support Vector Regression with Polynomial Kernel (SVR\_POL)

### Inputs

Table 4.27: Table of parameters specific to all support vector regression algorithms that use **MATLAB's built-in regressor (SVR\_POL)**.

Parameter	Format	Default	Description
SVR.parallel	binary	1	1: Use parallel computation; otherwise 0.
SVR.tune	binary	0	1: Tune $\epsilon$ automatically using cross-validation; otherwise 0.
SVR.KFold	vector/integer	5	When <code>SVR.tune = 1</code> , specifies the number of folds used in cross-validation.
SVR.default	binary	1	1: Use the default $\epsilon$ , i.e., <code>iqr(Y)/1.349</code> ; otherwise 0.
SVR.epsilon	vector/float	1e-4	A predefined $\epsilon$ (defines the margin within which errors are discounted from the SVR loss) when <code>opt.SVR.tune</code> and <code>opt.SVR.default</code> are both 0.

### Tips

- Unless the predictor data set is very large, data normalization is recommended.
- When `SVR.tune = 1`, cross-validation is conducted among log-scaled values in the range `[1e-3, 1e2]*iqr(Y)/1.349`, where `iqr(Y)/1.349` is an estimate of one-tenth of the standard deviation, using the interquartile range of the response variable **Y**. Note that computational times are generally very long when conducting CV here.
- The built-in MATLAB function `fitrsvm` used for this subgroup of methods applies the sequential minimal optimization routine by default to solve the dual problem of the support vector regression. See Smola and Schölkopf [58] for more details.

### Examples

```
>> model = daline.fit(data, 'method.name', 'SVR_POL');
```

```
>> model = daline.fit(data, 'method.name', 'SVR_POL', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'SVR.default', 0, 'SVR.epsilon', 0.05);
```

```
>> opt = daline.setopt('method.name', 'SVR_POL', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'SVR.default', 0, 'SVR.epsilon', 0.05);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_SVR_POL(data, 'variable.predictor', {'P', 'Q'}, 'variable.
      response', {'PF', 'Vm'}, 'SVR.default', 0, 'SVR.epsilon', 0.05);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
      }, 'SVR.default', 0, 'SVR.epsilon', 0.05);
>> model = func_algorithm_SVR_POL(data, opt);
```

## More About

In kernel-based support vector regression, each realization of  $\mathbf{x}$ , i.e.,  $\mathbf{x}_i$ , is first projected to a  $N_\phi$ -dimensional space via the mapping function  $\phi(\mathbf{x}_i)$ . The support vector regression model then becomes

$$\begin{aligned} \min_{\boldsymbol{\beta}_\phi, \xi_{ij}, \xi_{ij}^*} \quad & \frac{1}{2} \|\boldsymbol{\beta}_\phi\|_2^2 + \omega \sum_{i=1}^{N_s} \sum_{j=1}^{N_y} (\xi_{ij} + \xi_{ij}^*) \\ \text{s.t.} \quad & y_{ij} - \phi(\mathbf{x}_i)^\top \boldsymbol{\beta}_{\phi j} \leq \epsilon + \xi_{ij}, \quad \forall i, j \\ & \phi(\mathbf{x}_i)^\top \boldsymbol{\beta}_{\phi j} - y_{ij} \leq \epsilon + \xi_{ij}^*, \quad \forall i, j \\ & \xi_{ij}, \xi_{ij}^* \geq 0, \quad \forall i, j \end{aligned}$$

where  $\boldsymbol{\beta}_{\phi j}$  refers to the  $j$ -th column of  $\boldsymbol{\beta}_\phi$ .

Calculation of the inner product  $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle \quad \forall i, j$  is necessary when solving the above problem. If using the polynomial kernel, the inner product is easily calculated as

$$\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle = h(\langle \mathbf{x}_i, \mathbf{x}_j \rangle) = (\mathbf{x}_i^\top \mathbf{x}_j + 1)^d$$

where  $h(\cdot)$  is a scalar function. SVR\_POL sets  $d = 3$  by default; i.e., a cubic kernel is used.

## References

Jiafan Yu, Yang Weng, and Ram Rajagopal. “Robust mapping rule estimation for power flow analysis in distribution grids”. In: *2017 North American Power Symposium (NAPS)*. IEEE. 2017, pp. 1–6

### 4.5.3 Support Vector Regression with Ridge Regression (SVR\_RR)

#### Additional inputs

Table 4.28: Table of parameters specific to support vector regression with ridge regression.

Parameter	Format	Default	Description
SVR.lambda	float	1e-10	The regularization factor for the ridge regression component.
SVR.PCA	binary	1	1: Use principal component analysis (PCA) to reduce data collinearity; otherwise 0.
SVR.numComponentRatio	float	70	Unit: %, i.e., the proportion of the number of principal components w.r.t. the number of predictors.

#### Tips

- The  $\|\beta\|_2^2$  term in SVR's objective function has a large coefficient of  $\frac{1}{2}$  by default so that the regressed function can be as flat as possible; i.e.,  $\beta$  is desired to be small. However, such regularization may regulate the variance too strongly and result in strong bias.
- In the current version of DALINE, only 'indivi' is supported as a valid argument for `SVR.programType`.
- It is highly recommended to use 'yalmip' as the language, 'quadprog' as the solver, and 'indivi' as the type of programming.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'SVR_RR');
```

```
>> model = daline.fit(data, 'method.name', 'SVR_RR', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'SVR.lambda', -0.4999, 'SVR.PCA', 0, 'SVR.language', 'yalmip', 'SVR.programType', 'indivi', 'SVR.solver', 'quadprog');
```

```
>> opt = daline.setopt('method.name', 'SVR_RR', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'SVR.lambda', -0.4999, 'SVR.PCA', 0);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_SVR_RR(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'SVR.lambda', -0.4999, 'SVR.PCA', 0);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'}, 'SVR.lambda', -0.4999, 'SVR.PCA', 0);
>> model = func_algorithm_SVR_RR(data, opt);
```

### More About

To confer additional control over the level of regularization performed in SVR, a tunable regularization term can be further introduced into the objective function as

$$\min_{\beta, \xi_{ij}, \xi_{ij}^*} \frac{1}{2} \|\beta\|_2^2 + \omega \sum_{i=1}^{N_s} \sum_{j=1}^{N_y} (\xi_{ij} + \xi_{ij}^*) + \lambda \|\beta\|_2^2$$

As done in [29], the option to perform PCA on the predictor matrix  $\mathbf{X}$  before fitting an SVR model is offered. See subsection 4.2.5 for more details.

### References

Penghua Li et al. “A Data-Driven Linear Optimal Power Flow Model for Distribution Networks”. In: *IEEE Transactions on Power Systems* (2022)

## 4.5.4 Support Vector Regression with Chance-constrained Programming (SVR\_CCP)

### Additional inputs

Table 4.29: Table of parameters specific to support vector regression with chance-constrained programming.

Parameter	Format	Default	Description
SVRCC.probThreshold	float	95	Probability threshold, $\zeta_j^{CCP}$ (as explained below), of the chance-constrained programming to solve SVR. Unit: %.
SVRCC.bigM	float	1e6	The value of big M when using chance-constrained programming to solve SVR.

### Tips

- The function parameters specified in Table 4.26 should use a prefix of **SVRCC** instead of **SVR**.
- It is recommended that **SVRCC.programType** be set to 'indivi', as the solution of an overall model usually requires long compute times and may cause MATLAB to hang. Naturally, the resultant optimal  $\hat{\beta}$  will also differ between the two approaches. Additionally, it is recommended to use 'yalmip' as the language and 'Gurobi' as the solver. Note that 'Gurobi' is an external, commercial solver that requires additional, separate installation procedure (this is one of the only two cases where 'Gurobi' is recommended in DALINE; see Section 2.2 for more details about the installation of 'Gurobi').

- Preliminary empirical testing suggests that, this method outperforms normal SVR in the presence of data outliers. In situations without outliers, `SVR_CC` may encounter difficulties with convergence when  $\epsilon$  is small; e.g.,  $1e-4$ . This issue can be circumvented by increasing the size of  $\epsilon$ ; e.g., to  $1e-3$ . However, in such cases where there are no outliers, simply using `SVR` is more accurate and faster because it does not use binary variables.
- Although this method avoids tuning  $\omega$ , it introduces another hyperparameter, the probability threshold  $\zeta_j^{CCP}$ . Thus, the challenge of setting hyperparameter values remains.
- Despite the  $j$  subscript, the current implementation of `SVR_CC` assigns a uniform probability threshold  $\zeta^{CCP}$  for all dependent variables in  $\mathbf{Y}$ .

## Examples

```
>> model = daline.fit(data, 'method.name', 'SVR_CC');
```

```
>> model = daline.fit(data, 'method.name', 'SVR_CC', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF', 'Vm'}, 'SVRCC.epsilon', 1e-3, 'SVRCC.probThreshold',
    0.95);
```

```
>> opt = daline.setopt('method.name', 'SVR_CC', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF', 'Vm'}, 'SVRCC.epsilon', 1e-3, 'SVRCC.probThreshold',
    0.95);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_SVR_CC(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'Vm'}, 'SVRCC.epsilon', 1e-3, 'SVRCC.probThreshold', 0.95);
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm'
    }, 'SVRCC.epsilon', 1e-3, 'SVRCC.probThreshold', 0.95);
>> model = func_algorithm_SVR_CC(data, opt);
```

## More About

Chance-constrained programming avoids the challenge of tuning  $\omega$  by removing the tolerance of residuals; i.e., the  $\omega \sum_{i=1}^{N_s} \sum_{j=1}^{N_y} (\xi_{ij} + \xi_{ij}^*)$  term in the SVR objective function, together with its corresponding constraints. Instead, single-chance constraints are added to restrict residuals, resulting in the model

$$\begin{aligned} \min_{\boldsymbol{\beta}} \quad & \frac{1}{2} \|\boldsymbol{\beta}\|_2^2 \\ \text{s.t.} \quad & \mathbb{P} \left\{ \|\mathbf{y}_j - \mathbf{x}^\top \boldsymbol{\beta}_j\|_1 \leq \epsilon \right\} \geq \zeta_j^{CCP}, \forall j \end{aligned}$$

where  $\mathbf{x} \in \mathbb{R}^{N_s \times N_x}$  is adopted over  $\mathbf{X}$  to highlight that  $\mathbf{X}$  is now  $N_s$  realisations of a multivariate random variable of dimension  $\mathbb{R}^{N_x}$ . In addition,  $\mathbf{y}_j \in \mathbb{R}$  refers to the  $j$ -th dependent variable in  $\mathbf{Y}$ ,

which should also be considered as  $N_s$  realisations of a multivariate random variable of dimension  $\mathbb{R}^{N_y}$ . The constant  $\zeta_{ij}^{CCP} \in \mathbb{R}$  is a predefined percentage threshold.

To solve the above problem, the big-M method is employed to convert it into a mixed-integer linear programming problem, based on the scenarios of  $\mathbf{y}_j$  and  $\mathbf{x}$ . The problem then takes the form

$$\begin{aligned}
& \min_{\boldsymbol{\beta}} \quad \frac{1}{2} \|\boldsymbol{\beta}\|_2^2 \\
& \text{s.t.} \quad \mathbf{y}_j - \mathbf{x}^\top \boldsymbol{\beta}_j \leq \epsilon + M(1 - z_j), \quad \forall j \\
& \quad \mathbf{x}^\top \boldsymbol{\beta}_j - \mathbf{y}_j \leq \epsilon + M(1 - z_j), \quad \forall j \\
& \quad \frac{1}{N_s} \sum_{i=1}^{N_s} z_{ij} \geq \zeta_j^{CCP}, \quad \forall j \\
& \quad z_{ij} \in \{0, 1\} \quad \forall i, j \\
& \quad M \gg \epsilon
\end{aligned}$$

where  $\mathbf{z}_j \in \mathbb{R}^{N_s \times 1}$  comprises of scalar elements  $z_{ij}$ .

In the case where `SVRCC.programType` is set to 'indivi', the constraints for only one value of  $j$  are implemented in the above optimization formulation. The problem is solved  $N_y$  times, once for each column in  $\mathbf{y}$ , and the resulting  $\hat{\boldsymbol{\beta}}$  from each solution are concatenated column-wise.

## References

Zhentong Shao et al. "A linear AC unit commitment formulation: An application of data-driven linear power flow model". In: *International Journal of Electrical Power & Energy Systems* 145 (2023), p. 108673

## 4.6 Linearly Constrained Programming Family

This class of functions covers the following approaches:

- Linearly constrained programming with box constraints (LCP\_BOX)
- Linearly constrained programming without box constraints (LCP\_BOXN)
- Linearly constrained programming with Jacobian guidance constraints, using unnormalized data (LCP\_JGD)
- Linearly constrained programming without Jacobian guidance constraints, using unnormalized data (LCP\_JGD)
- Linearly constrained programming with coupling constraints (LCP\_COU, LCP\_COU2)
- Linearly constrained programming without coupling constraints (LCP\_COUN, LCP\_COUN2)

### 4.6.1 General Inputs

Table 4.30: Table of parameters common to all linearly constrained programming approaches.

Parameter	Format	Default	Description
LCP.solver	character	'Mosek'	Solver options; choose amongst 'quadprog', 'Gurobi', 'SDPT3' and 'SeDuMi' ('SDPT3' and 'SeDuMi' are included in DALINE via CVX; 'quadprog' is built in the MATLAB Optimization Toolbox; for 'Gurobi', you need to install it manually). For LCP_JGD and LCP_JGDN, the solver is fixed to be 'SDPT3', without which solutions cannot be found.
LCP.cvxQuiet	binary	1	1: Suppress CVX output in the command window; otherwise 0.

### 4.6.2 General Tips

- In the event that a program is interrupted while CVX is midway through solving, input the following codes into the MATLAB command window to shut down the CVX process. This will prevent future errors in calling/selecting the CVX solver, e.g., "The global CVX solver selection cannot be changed while a model is being constructed."

```
>> cvx_begin;
>> cvx_end;
```

### 4.6.3 Linearly Constrained Programming with Box Constraints (LCP\_BOX)

#### Tips

- This method fixes the predictors as **P**, **Q** and the responses as **Vm**, **Va** in **variable.predictor** and **variable.response** respectively. Changing the inputs to these function arguments will have no

effect on the output. Note that this also means that the known voltages of PV buses is not used in the DPFL model training, leading to a possibly significant loss of information.

- It is recommended to use 'SeDuMi' as the solver.

## Examples

```
>> model = daline.fit(data, 'method.name', 'LCP_BOX');
```

```
>> model = daline.fit(data, 'method.name', 'LCP_BOX', 'LCP.solver', 'SeDuMi', 'LCP.
    cvxQuiet', 1);
```

```
>> opt = daline.setopt('method.name', 'LCP_BOX', 'LCP.solver', 'SeDuMi', 'LCP.cvxQuiet',
    0);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LCP_BOX(data, 'LCP.solver', 'SeDuMi', 'LCP.cvxQuiet', 0);
```

```
>> opt = daline.setopt('LCP.solver', 'SeDuMi', 'LCP.cvxQuiet', 0);
>> model = func_algorithm_LCP_BOX(data, opt);
```

## More About

This method solves an optimization problem with the objective function  $\min_{\beta} \|Y - X\beta\|_2^2$  and box constraints. It relies on computation of the Jacobian matrix

$$J = \frac{dx}{dy}$$

where  $\mathbf{x}$  refers to the active and reactive power injections, and  $\mathbf{y}$  to voltage magnitudes and angles, of a single observation. At around operating point 0,

$$J_0 = \left. \frac{dx}{dy} \right|_0 \approx \frac{\mathbf{x} - \mathbf{x}_0}{\mathbf{y} - \mathbf{y}_0}$$

A first-order Taylor approximation around  $\mathbf{y}$  can therefore be derived as

$$\mathbf{y} \approx \mathbf{y}_0 + J_0^{-1} \mathbf{x} - J_0^{-1} \mathbf{x}_0$$



which can also be represented as

$$\mathbf{y}^\top = \begin{bmatrix} 1 & \mathbf{x}^\top \end{bmatrix} \underbrace{\begin{bmatrix} \mathbf{y}_0^\top - \mathbf{x}_0^\top \mathbf{J}_0^{-1\top} \\ \mathbf{J}_0^{-1\top} \end{bmatrix}}_{\tilde{\boldsymbol{\beta}}}$$

for each data point  $\mathbf{x}$ .

Consequently,  $\tilde{\boldsymbol{\beta}}$  can be used as box constraints for  $\boldsymbol{\beta}$  in a linearly-constrained program; i.e.,

$$\tilde{\boldsymbol{\beta}}_{\min} \leq \boldsymbol{\beta} \leq \tilde{\boldsymbol{\beta}}_{\max}$$

when many samples around the operating point 0 have been taken.

## References

Yuxiao Liu et al. “Bounding regression errors in data-driven power grid steady-state models”. In: *IEEE Transactions on Power Systems* 36.2 (2020), pp. 1023–1033

### 4.6.4 Linearly Constrained Programming without Box Constraints (LCP\_BOXN)

#### Tips

- This function should be used for comparison with LCP\_BOXN to examine the effects of adding box constraints.
- See also the tips in Section 4.6.3.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'LCP_BOXN');
```

```
>> model = daline.fit(data, 'method.name', 'LCP_BOXN', 'LCP.solver', 'SeDuMi', 'LCP.
    cvxQuiet', 1);
```

```
>> opt = daline.setopt('method.name', 'LCP_BOXN', 'LCP.solver', 'SeDuMi', 'LCP.cvxQuiet', 0);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LCP_BOXN(data, 'LCP.solver', 'SeDuMi', 'LCP.cvxQuiet', 0);
```

```
>> opt = daline.setopt('LCP.solver', 'SeDuMi', 'LCP.cvxQuiet', 0);
>> model = func_algorithm_LCP_BOXN(data, opt);
```

### More About

This method solves an optimization problem with the objective function  $\min_{\beta} \|Y - X\beta\|_2^2$  and no constraints.

### References

Yuxiao Liu et al. “Bounding regression errors in data-driven power grid steady-state models”. In: *IEEE Transactions on Power Systems* 36.2 (2020), pp. 1023–1033

## 4.6.5 Linearly Constrained Programming with Jacobian Guidance Constraints (LCP\_JGD)

### Tips

- This method fixes the predictors as  $P$ ,  $Q$  and the responses as  $V_m$ ,  $V_a$  in `variable.predictor` and `variable.response` respectively. Changing the inputs to these function arguments will have no effect on the output. The predictors are denoted by  $Y$  and the responses by  $X$  in the section detailing LCP\_JGD below. However, note that there are known and unknown variables in  $Y$  and  $X$ ; i.e., not all “predictors” are known and not all “responses” are unknown, as LCP\_JGD uses the bundling strategy for known and unknown variables; see Section 4.3.4 for more details of this bundling strategy.
- To improve solving success rates, using unnormalized data for training is strongly recommended.

### Examples

```
>> model = daline.fit(data, 'method.name', 'LCP_JGD');
```

```
>> model = daline.fit(data, 'method.name', 'LCP_JGD', 'LCP.cvxQuiet', 0);
```

```
>> opt = daline.setopt('method.name', 'LCP_JGD', 'LCP.cvxQuiet', 0);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LCP_JGD(data, 'LCP.cvxQuiet', 0);
```

```
>> opt = daline.setopt('LCP.cvxQuiet', 0);
>> model = func_algorithm_LCP_JGD(data, opt);
```

### More About

This method solves the LCQP

$$\begin{aligned} \min_{\substack{\beta, \mathbf{C}, \\ \beta_B, \beta_G, \beta_P, \beta_Q}} \quad & \|\mathbf{Y} - \mathbf{X}\beta^\top - \mathbf{C}\|_2^2 \\ \text{s.t.} \quad & \beta = \begin{bmatrix} \beta_B & -\beta_G \\ \beta_G & \beta_B \end{bmatrix} + \begin{bmatrix} -\beta_Q & \beta_P \\ \beta_P & \beta_Q \end{bmatrix} \end{aligned}$$

where  $\beta$  is square and of dimension equal to the number of voltage and voltage angle responses,  $\beta_B$  and  $\beta_G$  are symmetric, and  $\beta_P$  and  $\beta_Q$  are diagonal matrices.  $\mathbf{C}$  is a matrix of constant terms and is included here because  $\beta$  does not contain a column of 1's under this formulation.

The constraints in the above problem are termed the Jacobian matrix-guided constraints, so named through the observation in [14] that the values of  $\beta$  are similar to those in the Jacobian matrix of the ACPF equations. By assuming that  $\sin \theta_{ij} \ll \cos \theta_{ij}$  holds in power systems, the Jacobian matrix can be simplified to achieve the symmetrical structure reflected in the constraint.

Finally, the bundling technique is applied to the known and unknown variables. By decomposing  $\beta$  and  $\mathbf{C}$ , the linearized PF equations can be expressed as

$$\begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{bmatrix} = \begin{bmatrix} \beta_{11} & \beta_{12} \\ \beta_{21} & \beta_{22} \end{bmatrix} \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix} + \begin{bmatrix} \mathbf{C}_1 \\ \mathbf{C}_2 \end{bmatrix}$$

in which  $\mathbf{Y}_1 = \begin{bmatrix} \mathbf{P}_{PQ}^\top & \mathbf{P}_{PV}^\top & \mathbf{Q}_{PQ}^\top \end{bmatrix}^\top$  and  $\mathbf{X}_2 = \begin{bmatrix} \boldsymbol{\theta}_{V\theta}^\top & \mathbf{V}_{PV}^\top & \mathbf{V}_{V\theta}^\top \end{bmatrix}^\top$  are known variables, while  $\mathbf{Y}_2 = \begin{bmatrix} \mathbf{P}_{V\theta}^\top & \mathbf{Q}_{PV}^\top & \mathbf{Q}_{V\theta}^\top \end{bmatrix}^\top$  and  $\mathbf{X}_1 = \begin{bmatrix} \boldsymbol{\theta}_{PQ}^\top & \boldsymbol{\theta}_{PV}^\top & \mathbf{V}_{PQ}^\top \end{bmatrix}^\top$  are unknown variables. With this, the unknown  $\hat{\mathbf{X}}_1$  and  $\mathbf{Y}_2$  can be calculated using the estimates from the LCQP as

$$\begin{aligned} \hat{\mathbf{X}}_1 &= \hat{\beta}_{11}^{-1} \left( \mathbf{Y}_1 - \hat{\beta}_{12} \mathbf{X}_2 - \hat{\mathbf{C}}_1 \right) \\ \hat{\mathbf{Y}}_2 &= \hat{\beta}_{21} \hat{\mathbf{X}}_1 + \hat{\beta}_{22} \mathbf{X}_2 + \hat{\mathbf{C}}_2 \end{aligned}$$

### References

Yuxiao Liu et al. "A data-driven approach to linearize power flow equations considering measurement noise". In: *IEEE Transactions on Smart Grid* 11.3 (2019), pp. 2576–2587

### 4.6.6 Linearly Constrained Programming without Jacobian Guidance Constraints (LCP\_JGDN)

#### Tips

- This function should be used for comparison with LCP\_JGD to examine the effects of adding Jacobian matrix-guided constraints.
- See also the tips in Section 4.6.5.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'LCP_JGDN');
```

```
>> model = daline.fit(data, 'method.name', 'LCP_JGDN', 'LCP.cvxQuiet', 0);
```

```
>> opt = daline.setopt('method.name', 'LCP_JGDN', 'LCP.cvxQuiet', 0);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LCP_JGDN(data, 'LCP.cvxQuiet', 0);
```

```
>> opt = daline.setopt('LCP.cvxQuiet', 0);
>> model = func_algorithm_LCP_JGDN(data, opt);
```

#### More About

This method solves the LCQP

$$\min_{\beta, C} \|Y - X\beta^\top - C\|_2^2 \quad (4.5)$$

without any additional constraints. For details on how the unknown variables in  $X$  and  $Y$  are found, see also the information in Section 4.6.5.

#### References

Yuxiao Liu et al. “A data-driven approach to linearize power flow equations considering measurement noise”. In: *IEEE Transactions on Smart Grid* 11.3 (2019), pp. 2576–2587

### 4.6.7 Linearly Constrained Programming with Coupling Constraints (LCP\_COU and LCP\_COU2)

#### Additional inputs

Table 4.31: Table of parameters specific to linearly constrained programming with coupling constraints.

Parameter	Format	Default	Description
<code>opt.idx</code>	function/struct	<code>func_define_idx</code>	A struct (or function calling a struct) that contains the column indices of the from- and to-buses in a MATPOWER-like <code>mpc.branch</code> matrix. By default, it uses the built-in index in MATPOWER.
<code>LCP.coupleDelta</code>	float	1e-2	A small non-negative value, $\delta$ , to define the bounds for the coupling constraints.

#### Tips

- This method fixes the predictors as  $V_a$ ,  $V_m$  (or  $V_a$ ,  $V_{m2}$  in the case of LCP\_COU2) and the responses as PF, PT, QF, QT in `opt.variable.predictor` and `opt.variable.response` respectively. Changing the inputs to these function arguments will have no effect on the output.
- LCP\_COU and LCP\_COU2 require a prior installation of the CVX optimization toolbox and the desired solvers (e.g., Gurobi, Mosek) to be used.
- The only difference between LCP\_COU and LCP\_COU2 is that LCP\_COU fixes  $V_a$ ,  $V_m$  as the predictors while LCP\_COU2 fixes  $V_a$ ,  $V_{m2}$  as the predictors. Hence, while the following examples are based on LCP\_COU, they are applicable to LCP\_COU2 as well.
- It is recommended to use 'quadprog' as the solver.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'LCP_COU');
```

```
>> model = daline.fit(data, 'method.name', 'LCP_COU', 'LCP.coupleDelta', 1e-3, 'LCP.solver', 'quadprog');
```

```
>> opt = daline.setopt('method.name', 'LCP_COU', 'LCP.coupleDelta', 1e-3, 'LCP.solver', 'quadprog');
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LCP_COU(data, 'LCP.coupleDelta', 1e-3, 'LCP.solver', 'quadprog');
```

```
>> opt = daline.setopt('LCP.coupleDelta', 1e-3, 'LCP.solver', 'quadprog');
>> model = func_algorithm_LCP_COU(data, opt);
```

### More About

This method exploits the strong correlation between certain elements in  $\beta$  arising from the physical dependencies within power systems. In the AC line flow model, voltage angles always appear in the form  $\theta_{nm} = \theta_n - \theta_m$ . Thus, when predicting power flows based on the voltages and angles of buses  $n$  and  $m$ , the coefficients in front of  $\theta_n$  and  $\theta_m$  in the DPFL model should be exactly opposite, i.e.,  $\beta_{nj} + \beta_{mj} = 0$ . This constraint can be slightly relaxed for practical use as

$$-\delta \leq \beta_{nj} + \beta_{mj} \leq \delta$$

where  $\delta \in \mathbb{R}$  is a pre-set, non-negative small value.

The resulting optimization problem is formulated as

$$\begin{aligned} \min_{\beta} \quad & \|Y - X\beta\|_2^2 \\ \text{s.t.} \quad & Y_j = \beta_{nj}\theta_n + \beta_{mj}\theta_m + \beta_{n'j}V_n + \beta_{m'j}V_m + \beta_{cj} \quad \forall j \in N_y \\ & -\delta \leq \beta_{nj} + \beta_{mj} \leq \delta \quad \forall j \in N_y \end{aligned}$$

where  $\theta_n$ ,  $\theta_m$ ,  $V_n$ , and  $V_m$  are the columns in  $X$  representing the voltage angles and magnitudes of buses  $n$  and  $m$ .  $n$  and  $m$  are thus the buses between which flows  $Y_j$ . In the case of LCP\_COU2, the squared voltages  $V_n^2$  and  $V_m^2$  are used in place of  $V_n$  and  $V_m$ .

### References

Yuxiao Liu et al. “Bounding regression errors in data-driven power grid steady-state models”. In: *IEEE Transactions on Power Systems* 36.2 (2020), pp. 1023–1033

### 4.6.8 Linearly Constrained Programming without Coupling Constraints (LCP\_COUN and LCP\_COUN2)

#### Additional inputs

Table 4.32: Table of parameters specific to linearly constrained programming without coupling constraints.

Parameter	Format	Default	Description
opt.idx	function/struct	func_define_idx	A struct (or function calling a struct) that contains the column indices of the from- and to-buses in a MATPOWER-like <code>mpc.branch</code> matrix. By default, it uses the built-in index in MATPOWER.

### Tips

- LCP\_COUN and LCP\_COUN2 should be used for comparison with LCP\_COU and LCP\_COU2 respectively, to examine the effects of adding coupling constraints.
- See also the tips in Section 4.6.7.
- The only difference between LCP\_COUN and LCP\_COUN2 is that LCP\_COUN fixes  $V_a$ ,  $V_m$  as the predictors while LCP\_COUN2 fixes  $V_a$ ,  $V_{m2}$  as the predictors. Hence, while the following examples are based on LCP\_COUN, they are applicable to LCP\_COUN2 as well.

### Examples

```
>> model = daline.fit(data, 'method.name', 'LCP_COUN');
```

```
>> model = daline.fit(data, 'method.name', 'LCP_COUN', 'LCP.solver', 'quadprog');
```

```
>> opt = daline.setopt('method.name', 'LCP_COUN', 'LCP.solver', 'quadprog');
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LCP_COUN(data, 'LCP.solver', 'quadprog');
```

```
>> opt = daline.setopt('LCP.solver', 'quadprog');
>> model = func_algorithm_LCP_COUN(data, opt);
```

### More About

This method solves the LCQP

$$\begin{aligned} \min_{\beta} \quad & \|Y - X\beta\|_2^2 \\ \text{s.t.} \quad & Y_j = \beta_{nj}\theta_n + \beta_{mj}\theta_m + \beta_{n'j}V_n + \beta_{m'j}V_m + \beta_{cj} \quad \forall j \in N_y \end{aligned}$$

where  $\theta_n$ ,  $\theta_m$ ,  $V_n$ , and  $V_m$  are the columns in  $X$  representing the voltage angles and magnitudes of buses  $n$  and  $m$ .  $n$  and  $m$  are thus the buses between which flows  $Y_j$ . In the case of LCP\_COUN2, the squared voltages  $V_n^2$  and  $V_m^2$  are used in place of  $V_n$  and  $V_m$ .

### References

Yuxiao Liu et al. “Bounding regression errors in data-driven power grid steady-state models”. In: *IEEE Transactions on Power Systems* 36.2 (2020), pp. 1023–1033

## 4.7 Distributionally Robust Chance-constrained Programming Family

This class of functions covers the following algorithms:

- Moment-based distributionally robust chance-constrained programming with  $\mathbf{X}$  as random variable (DRC\_XM)
- Moment-based distributionally robust chance-constrained programming with  $\mathbf{X}$  and  $\mathbf{Y}$  as random variables (DRC\_XYM)
- Divergence-based distributionally robust chance-constrained programming with  $\mathbf{X}$  and  $\mathbf{Y}$  as random variables (DRC\_XYD)

### General Inputs

Table 4.33: Table of parameters common to all distributionally robust chance-constrained programming approaches.

Parameter	Format	Default	Description
DRC.programType	character	'indivi'	'whole' puts all responses (i.e., the number of columns in $\mathbf{Y}$ ) into one optimization program to solve for all $\hat{\beta}$ at once; 'indivi' solves for $\hat{\beta}$ individually by building one optimization program for each response.
DRC.epsilon	float	1e-4	The threshold, $\epsilon$ , within which residuals must fall in the distributionally robust chance constraint.
DRC.probThreshold	float	95	Unit: %, i.e., the probability threshold $\zeta^{DRC}$ of the distributionally robust chance constraint. Must take a value from 0 to 100.
DRC.starIDX	character/integer	'end'	Index of the operating point of interest in the training dataset; should be less than or equal to <code>opt.num.trainSample</code> , e.g., <code>DRC.starIDX = 200</code> when <code>opt.num.trainSample</code> is 300. If its argument is 'end', the method uses the last sample in the training dataset as the operating point of interest.
DRC.language	character	'yalmip'	Optimization toolbox to formulate the programming problem. Choose between 'cvx' or 'yalmip'.
DRC.parallel	binary	1	1: Use parallel computation; otherwise 0. Only valid when <code>DRC.language = 'yalmip'</code> and <code>DRC.programType = 'indivi'</code> , because to the best of the developers' knowledge, <code>cvx</code> does not support parallel computing.
DRC.cvxQuiet	binary	1	1: Suppress CVX output in the command window; otherwise 0.
DRC.yalDisplay	binary	0	1: Show YALMIP display; otherwise 0.

### General Tips

- The value of `DRC.epsilon` greatly affects the accuracy and feasibility of the optimization program: decreasing the value of  $\epsilon$  creates tighter constraints but may result in there being no solution(s).
- The selection of `DRC.epsilon` is highly influenced by the scale of the data. When data are normalized, a smaller `DRC.epsilon` value is typically more appropriate compared to when using data in its original scale. Furthermore, employing a very small `DRC.epsilon` with data on its original scale can often lead to linear models that exhibit significantly large errors.



- The functions in this class require a prior installation of the CVX optimization toolbox and the desired solvers ( e.g., Gurobi, Mosek) to be used.
- The design of these methods adopt the common idea: train a linear power flow model around a designated operating point  $(\mathbf{y}^*, \mathbf{x}^*)$ . Hence, the resulting model is most accurate when the test data set has an operating state close to  $(\mathbf{y}^*, \mathbf{x}^*)$ .
- In the event that a program is interrupted while CVX is midway through solving, input `cvx_begin` and `cvx_end` into the MATLAB command window to shut down the CVX process. This will prevent future errors in calling the CVX solver.

```
>> cvx_begin;
>> cvx_end;
```

### More About

The distributionally robust chance-constrained programs are formulated as

$$\begin{aligned} \min_{\beta} \quad & \|\mathbf{y}^{\star\top} - \mathbf{x}^{\star\top} \beta\|_2^2 \\ \text{s.t.} \quad & \inf_{\mathbb{P}(\mathcal{X}) \in \mathcal{D}} \mathbb{P} \{ \tilde{r}_j(\mathcal{Y}_j, \mathcal{X}, \beta_j) \leq \epsilon \} \geq \zeta^{DRC} \quad \forall j \end{aligned}$$

where  $r(\cdot)$  is a general description of a residual, and  $\mathbf{y}^* \in \mathbb{R}^{N_y \times 1}$  and  $\mathbf{x}^* \in \mathbb{R}^{N_x \times 1}$  are two given realizations of  $\mathbf{y}$  and  $\mathbf{x}$  that together represent a typical operating point.  $\mathcal{X}$  and  $\mathcal{Y}_j$  are random variables referring to a row in  $\mathbf{X}$  and the  $j$ -th variable in a row in  $\mathbf{Y}$  respectively, while  $\epsilon \in \mathbb{R}$  and  $\zeta^{DRC} \in \mathbb{R}$  are two preset thresholds.

Since it is often difficult to accurately assume a prior probability distribution for random variables, an ambiguity set  $\mathcal{D}$  is utilized to describe multiple possible selections of  $\mathbb{P}(\mathcal{X})$ , yielding the distributionally robust chance constraint shown above. In this constraint,  $\tilde{r}_j(\cdot)$  is defined as a linear approximation of the original residuals  $r(\mathbf{y}^*, \mathbf{x}^*, \beta) = \|\mathbf{y}^{\star\top} - \mathbf{x}^{\star\top} \beta\|_2^2$ , and is formulated as

$$\tilde{r}_j(\mathcal{Y}_j, \mathcal{X}, \beta_j) = \|\mathcal{Y}_j - \mathcal{X}^\top \beta_j\|_1, \quad \forall j$$

To solve the DRCC problem, the distributionally robust chance constraints are reformulated. If the ambiguity set  $\mathcal{D}$  is moment-based, the constraints can be equivalently transformed into semi-definite constraints via the conic dual transformation, as done in `DRC_XM` and `DRC_XYM`. Alternatively, if  $\mathcal{D}$  is  $\phi$ -divergence-based, the distributionally robust chance constraints can be mapped to traditional single-chance constraints, yielding `DRC_XYD`.

### 4.7.1 Moment-based Distributionally Robust Chance-constrained Programming with $\mathbf{X}$ as Random Variable (DRC\_XM)

#### Additional inputs

Table 4.34: Table of parameters specific to moment-based distributionally robust chance-constrained programming with  $\mathbf{X}$  as random variable.

Parameter	Format	Default	Description
DRC.gamma1	float	0	$\gamma_1$ , a SDP parameter reflecting the decision maker's risk preference. Default value taken from [66].
DRC.gamma2	float	1	$\gamma_2$ , a SDP parameter reflecting the decision maker's risk preference. Default value taken from [66].
DRC.solverM	character	'SeDuMi'	Solver options: 'SDPT3', 'SeDuMi' ('SDPT3' and 'SeDuMi' are included in DALINE via CVX).

#### Tips

- $\gamma_1$  and  $\gamma_2$  can be chosen based on the size of the training data set, respective confidence levels of  $\mu_0$  and  $\Sigma_0$ , and risk preference. Larger values of  $\gamma_1$  and  $\gamma_2$  generally result in more robust (conservative) solutions.
- DRC\_XM has very long compute times, especially when DRC.programType is set to 'whole', and may cause MATLAB to hang. Use of DRC\_XYM or, better, DRC\_XYD is recommended instead.
- It is recommended to use 'cvx' as the language and 'SeDuMi' as the solver.

#### Examples

```
>> data = daline.normalize(data);
>> model = daline.fit(data, 'method.name', 'DRC_XM');
```

```
>> data = daline.normalize(data);
>> model = daline.fit(dataN, 'method.name', 'DRC_XM', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF'}, 'DRC.epsilon', 1e-3, 'DRC.language', 'cvx', 'DRC.
    solverM', 'SeDuMi', 'DRC.programType', 'indivi');
```

```
>> data = daline.normalize(data);
>> opt = daline.setopt('method.name', 'DRC_XM', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'}, 'DRC.epsilon', 1e-3, 'DRC.language', 'cvx', 'DRC.solverM
    ', 'SeDuMi');
>> model = daline.fit(dataN, opt);
```

```
>> data = daline.normalize(data);
>> model = func_algorithm_DRC_XM(dataN, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF'}, 'DRC.epsilon', 1e-3, 'DRC.language', 'cvx', 'DRC.solverM', 'SeDuMi
    ');
```

```
>> data = daline.normalize(data);
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'}, '
    DRC.epsilon', 1e-3, 'DRC.language', 'cvx', 'DRC.solverM', 'SeDuMi');
>> model = func_algorithm_DRC_XM(dataN, opt);
```

### More About

In this method, the ambiguity set  $\mathcal{D}$  is constructed as a moment-based ambiguity set using the mean vector  $\boldsymbol{\mu}_0 = \frac{1}{N_s} \sum_{i=1}^{N_s} \mathcal{X}_i$  and covariance matrix  $\boldsymbol{\Sigma}_0 = \frac{1}{N_s} \sum_{i=1}^{N_s} (\mathcal{X}_i - \boldsymbol{\mu}_0)(\mathcal{X}_i - \boldsymbol{\mu}_0)^\top$ . The distributionally robust chance constraints

$$\inf_{\mathbb{P}(\mathcal{X}) \in \mathcal{D}_j} \mathbb{P} \left\{ \|\mathcal{Y}_j - \mathcal{X}^\top \beta_j\|_1 \leq \epsilon \right\} \geq \zeta^{DRC}, \quad \forall j$$

are then each reformulated into

$$\begin{aligned} & \gamma_2 \boldsymbol{\Sigma}_0 \cdot \mathbf{G}_j + 1 - r_j + \boldsymbol{\Sigma}_0 \cdot \mathbf{H}_j + \gamma_1 q_j \leq \zeta^{DRC} \lambda_j, \\ & \begin{bmatrix} \mathbf{G}_i & -\mathbf{p}_j \\ -\mathbf{p}_j^\top & 1 - r_j \end{bmatrix} \geq \begin{bmatrix} \mathbf{0} & -\frac{1}{2}\beta_j \\ -\frac{1}{2}\beta_j^\top & \lambda_j - \beta_j^\top \boldsymbol{\mu}_0 - \epsilon + \mathbf{y}^* \end{bmatrix}, \\ & \begin{bmatrix} \mathbf{G}_i & -\mathbf{p}_j \\ -\mathbf{p}_j^\top & 1 - r_j \end{bmatrix} \geq \begin{bmatrix} \mathbf{0} & \frac{1}{2}\beta_j \\ \frac{1}{2}\beta_j^\top & \lambda_j + \beta_j^\top \boldsymbol{\mu}_0 - \epsilon - \mathbf{y}^* \end{bmatrix}, \\ & \begin{bmatrix} \mathbf{G}_i & -\mathbf{p}_j \\ -\mathbf{p}_j^\top & 1 - r_j \end{bmatrix} \geq 0, \quad \begin{bmatrix} \mathbf{H}_i & \mathbf{p}_j \\ \mathbf{p}_j^\top & q_j \end{bmatrix} \geq 0, \quad \lambda_j \geq 0 \end{aligned}$$

where the operation  $\mathbf{A} \cdot \mathbf{B}$  denotes the trace of  $\mathbf{AB}$ ;  $\mathbf{G}_j$ ,  $\mathbf{H}_j$ ,  $r_j$ ,  $\mathbf{p}_j$ , and  $q_j$  are dual variables of the dual problem; and  $\lambda_j$  is a non-negative decision variable that enables the constraints to be finite SDP instead of semi-infinite. The two sets of SDP constraints represented by the second and third inequalities arise from that  $\mathbb{P}(\mathcal{Y}_j - \mathcal{X}^\top \beta_j \leq \epsilon)$  and  $\mathbb{P}(\mathcal{Y}_j - \mathcal{X}^\top \beta_j \geq -\epsilon)$  must be satisfied simultaneously. For detailed derivations, refer to [67].  $\gamma_1$ ,  $\gamma_2$ ,  $\beta$ ,  $\epsilon$ ,  $\zeta^{DRC}$  are as described in the “More About” subsection for this class of functions.

In the default case where `DRC.programType` is set to 'indivi', the optimization program is solved  $N_y$  times, once for each  $\mathcal{Y}_j$ , and the resulting  $\hat{\beta}$  from each solution are concatenated column-wise. If `DRC.programType` is instead set to 'whole', the program is run once using  $\mathcal{Y}$  and  $\beta$  in the distributionally robust chance constraints.

## References

- Yitong Liu, Zhengshuo Li, and Junbo Zhao. “Robust Data-Driven Linear Power Flow Model With Probability Constrained Worst-Case Errors”. In: *IEEE Transactions on Power Systems* 37.5 (2022), pp. 4113–4116. DOI: [10.1109/TPWRS.2022.3189543](https://doi.org/10.1109/TPWRS.2022.3189543)
- Yiling Zhang, Siqian Shen, and Johanna L Mathieu. “Distributionally robust chance-constrained optimal power flow with uncertain renewables and uncertain reserves provided by loads”. In: *IEEE Transactions on Power Systems* 32.2 (2016), pp. 1378–1388

## 4.7.2 Moment-based Distributionally Robust Chance-constrained Programming with $X$ and $Y$ as Random Variables (DRC\_XYM)

### Additional inputs

Refer to the additional parameters listed in Table 4.34.

### Tips

- See the tips in Section 4.7.1.

### Examples

```
>> data = daline.normalize(data);
>> model = daline.fit(data, 'method.name', 'DRC_XYM');
```

```
>> data = daline.normalize(data);
>> model = daline.fit(dataN, 'method.name', 'DRC_XYM', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF'}, 'DRC.probThreshold', 90, 'DRC.gamma2', 0.5, 'DRC.
    language', 'cvx', 'DRC.solverM', 'SeDuMi', 'DRC.programType', 'whole');
```

```
>> data = daline.normalize(data);
>> opt = daline.setopt('method.name', 'DRC_XYM', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'}, 'DRC.probThreshold', 90, 'DRC.gamma2', 0.5);
>> model = daline.fit(dataN, opt);
```

```
>> data = daline.normalize(data);
>> model = func_algorithm_DRC_XYM(dataN, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF'}, 'DRC.probThreshold', 90, 'DRC.gamma2', 0.5);
```

```
>> data = daline.normalize(data);
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'}, '
    DRC.probThreshold', 90, 'DRC.gamma2', 0.5);
>> model = func_algorithm_DRC_XYM(dataN, opt);
```

### More About

The model presented in `func_algorithm_DRC_XM` treats only  $\mathcal{X}$  as a stochastic parameter, and thus calculates  $\mathbb{P}(\mathcal{X})$  but not  $\mathbb{P}(\mathcal{Y}_j)$ . In contrast, the method here builds upon the joint probability distribution,  $\mathbb{P}(\mathcal{X}, \mathcal{Y}_j)$  for all  $j$ .

For each ambiguity set  $\mathcal{D}_j$  constructed, define  $\tilde{\mathcal{X}}_j$  as  $\tilde{\mathcal{X}}_j = \begin{bmatrix} \mathcal{Y}_j & \mathcal{X} \end{bmatrix}$ , such that  $\mu_j = \frac{1}{N_s+1} \sum_{i=1}^{N_s+1} \tilde{\mathcal{X}}_{j,i}$  and covariance matrix  $\Sigma_j = \frac{1}{N_s+1} \sum_{i=1}^{N_s+1} (\tilde{\mathcal{X}}_{j,i} - \mu_j)(\tilde{\mathcal{X}}_{j,i} - \mu_j)^\top$ .

The distributionally robust chance constraints

$$\inf_{\mathbb{P}(\mathcal{X}) \in \mathcal{D}} \mathbb{P} \left\{ \|\mathcal{Y}_j - \mathcal{X}^\top \beta_j\|_1 \leq \epsilon \right\} \geq \zeta^{DRC}, \quad \forall j$$

are then each reformulated into

$$\begin{aligned} \gamma_2 \Sigma_j \cdot \mathbf{G}_j + 1 - r_j + \Sigma_j \cdot \mathbf{H}_j + \gamma_1 q_j &\leq \zeta^{DRC} \lambda_j, \\ \begin{bmatrix} \mathbf{G}_i & -\mathbf{p}_j \\ -\mathbf{p}_j^\top & 1 - r_j \end{bmatrix} &\geq \begin{bmatrix} \mathbf{0} & -\frac{1}{2} \beta_j' \\ -\frac{1}{2} \beta_j'^\top & \lambda_j - \beta_j'^\top \mu_j - \epsilon \end{bmatrix}, \\ \begin{bmatrix} \mathbf{G}_i & -\mathbf{p}_j \\ -\mathbf{p}_j^\top & 1 - r_j \end{bmatrix} &\geq \begin{bmatrix} \mathbf{0} & \frac{1}{2} \beta_j'' \\ \frac{1}{2} \beta_j''^\top & \lambda_j + \beta_j''^\top \mu_j - \epsilon \end{bmatrix}, \\ \begin{bmatrix} \mathbf{G}_i & -\mathbf{p}_j \\ -\mathbf{p}_j^\top & 1 - r_j \end{bmatrix} &\geq 0, \quad \begin{bmatrix} \mathbf{H}_i & \mathbf{p}_j \\ \mathbf{p}_j^\top & q_j \end{bmatrix} \geq 0, \quad \lambda_j \geq 0 \end{aligned}$$

where the operation  $\mathbf{A} \cdot \mathbf{B}$  denotes the trace of  $\mathbf{AB}$ ;  $\mathbf{G}_j$ ,  $\mathbf{H}_j$ ,  $r_j$ ,  $\mathbf{p}_j$ , and  $\mathbf{q}_j$  are dual variables of the dual problem;  $\lambda_j$  is a non-negative decision variable;  $\beta_j' = \begin{bmatrix} -1 & \beta_j^\top \end{bmatrix}^\top$  and  $\beta_j'' = \begin{bmatrix} 1 & \beta_j^\top \end{bmatrix}^\top$ . Finally,  $\gamma_1$ ,  $\gamma_2$ ,  $\beta$ ,  $\epsilon$ ,  $\zeta^{DRC}$  are as described in the “More About” subsection for this class of functions.

In the default case where `DRC.programType` is set to 'indivi', the optimization program is solved

$N_y$  times, once for each  $\mathbf{y}_j$ , and the resulting  $\hat{\beta}$  from each solution are concatenated column-wise. If `DRC.programType` is instead set to 'whole', the program is run once using  $\mathbf{y}$  and  $\beta$  in the distributionally robust chance constraints.

## References

- Yitong Liu, Zhengshuo Li, and Junbo Zhao. “Robust Data-Driven Linear Power Flow Model With Probability Constrained Worst-Case Errors”. In: *IEEE Transactions on Power Systems* 37.5 (2022), pp. 4113–4116. DOI: [10.1109/TPWRS.2022.3189543](https://doi.org/10.1109/TPWRS.2022.3189543)
- Yiling Zhang, Siqian Shen, and Johanna L Mathieu. “Distributionally robust chance-constrained optimal power flow with uncertain renewables and uncertain reserves provided by loads”. In: *IEEE Transactions on Power Systems* 32.2 (2016), pp. 1378–1388

### 4.7.3 Divergence-based Distributionally Robust Chance-Constrained Programming with $\mathbf{X}$ and $\mathbf{Y}$ as Random Variables (DRC\_XYD)

#### Additional inputs

Table 4.35: Table of parameters specific to divergence-based distributionally robust chance-constrained programming with  $\mathbf{X}$  and  $\mathbf{Y}$  as random variables.

Parameter	Format	Default	Description
<code>DRC.d</code>	float	0.1	Tolerance, $d$ , of the distance between the particular density function and the reference one. Default value taken from Liu, Li, and Zhao [68].
<code>DRC.infMethod</code>	character	'fzero'	Choose between 'fzero' or 'bisec'. These refer to the methods (MATLAB's built-in fzero function and bisection search) of finding the infimum of $h(x)$ for ambiguity sets that use KL divergence.
<code>DRC.delta</code>	float	1e-10	A very small value, $\delta$ , to form an exclusive interval $[\delta, 1 - \delta]$ within $(0, 1)$ . Relevant only if <code>DRC.infMethod</code> is set to 'fzero'.
<code>DRC.bisecTol</code>	float	1e-9	Tolerance of error for the bisection search; should be very small. Relevant only if <code>DRC.infMethod</code> is set to 'bisec'.
<code>DRC.bigM</code>	float	1e6	Value for the big $M$ .
<code>DRC.solverD</code>	character	'Gurobi'	Solver option. Only 'Gurobi' is suggested here (for 'Gurobi', you need to install it manually).

#### Tips

- The results returned by `DRC_XYD` are usually identical whether using `fzero` or the bisection search to find the infimum of  $h(x)$ . However, if either method fails, it is worthwhile to try using the other.
- The 'whole' option is currently not available as a working argument to `DRC.programType`.

- With DRC\_XYD, using YALMIP is often faster than using CVX.
- It is recommended to use 'Gurobi' as the solver. Note that 'Gurobi' is an external, commercial solver that requires additional, separate installation procedure (this is one of the only two cases where 'Gurobi' is recommended in DALINE; see Section 2.2 for more details about the installation of 'Gurobi')
- Although computationally faster than DRC\_XM and DRC\_XYM, the solution time of DRC\_XYD is generally still significantly longer than those of other regression-based DPFL approaches.

## Examples

```
>> data = daline.normalize(data);
>> model = daline.fit(data, 'method.name', 'DRC_XYD');
```

```
>> data = daline.normalize(data);
>> model = daline.fit(dataN, 'method.name', 'DRC_XYD', 'variable.predictor', {'P', 'Q'},
    'variable.response', {'PF'}, 'DRC.infMethod', 'bisec', 'DRC.bisecTol', 1e-8, 'DRC.
    language', 'yalmip', 'DRC.solverD', 'Gurobi', 'DRC.programType', 'indivi');
```

```
>> data = daline.normalize(data);
>> opt = daline.setopt('method.name', 'DRC_XYD', 'variable.predictor', {'P', 'Q'}, '
    variable.response', {'PF'}, 'DRC.infMethod', 'bisec', 'DRC.bisecTol', 1e-8);
>> model = daline.fit(dataN, opt);
```

```
>> data = daline.normalize(data);
>> model = func_algorithm_DRC_XYD(dataN, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF'}, 'DRC.infMethod', 'bisec', 'DRC.bisecTol', 1e-8);
```

```
>> data = daline.normalize(data);
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'}, '
    DRC.infMethod', 'bisec', 'DRC.bisecTol', 1e-8);
>> model = func_algorithm_DRC_XYD(dataN, opt);
```

## More About

In this method, the ambiguity set  $\mathcal{D}$  is built around a reference distribution for  $\mathcal{X}$ ; it is assumed that the true density function is near this reference.  $\phi$ -divergence,  $D_\phi(f\|f_0)$ , is then used to describe the distance between a particular probability density function  $f$  and the reference's probability density function  $f_0$ .

As discussed in Theorem C.1 of [69], the distributionally robust chance constraint

$$\inf_{\mathbb{P}(\mathbf{x}) \in \{D_\phi(f\|f_0) \leq d\}} \mathbb{P}\{\tilde{r}_j(\mathcal{Y}_j, \mathcal{X}, \beta_j) \leq \epsilon\} \geq \zeta^{DRC} = 1 - \alpha$$

where  $d$  is the tolerance of the distance between the particular and the reference density functions, and  $\alpha$  is the maximum allowed probability of constraint violation, is equivalent to the chance constraint

$$\mathbb{P}_0 \{ \tilde{r}_j(\mathbf{y}_j, \mathbf{x}, \boldsymbol{\beta}_j) \leq \epsilon \} \geq 1 - \alpha'_+$$

where  $\mathbb{P}_0$  indicates probability evaluated at the reference distribution and  $\alpha'_+ = \max\{\alpha', 0\}$ . `DRC_XYD` uses Kullback-Leibler (KL) divergence as its  $\phi$ -divergence, such that

$$\alpha' = 1 - \inf_{x \in (0,1)} \underbrace{\left\{ \frac{e^{-d}x^{1-\alpha} - 1}{x - 1} \right\}}_{h(x)}$$

For  $x \in (0,1)$ , it can be shown that the convex function  $h(x)$  has an infimum that is attainable. `func_algorithm_DRC_XYD` calculates this infimum by finding the root of  $\frac{\partial h}{\partial x}$ , either with MATLAB's `fsolve` function or via the bisection method. With the former, `DRC.delta` can be adjusted to determine the interval  $[\delta, 1-\delta]$  within which `fsolve` will search for a root. With the latter, `DRC.bisecTol` determines the maximum interval size before the bisection search stops; i.e., it is the error tolerance for the bisection search.

Instead of ascertaining a suitable reference distribution, the chance constraints can be further transformed into deterministic ones using the big-M approach, as performed in [26]. For each response in  $\mathbf{y}$ , the final optimization program is thus formulated as

$$\begin{aligned} & \min_{\boldsymbol{\beta}_j, \mathbf{z}_j} \|\mathbf{y}_j^* - \mathbf{x}^{*\top} \boldsymbol{\beta}_j\|_2^2 \\ \text{s.t. } & \mathbf{y}_{ij} - \mathbf{x}_i \boldsymbol{\beta}_j \leq \epsilon z_{ij} + M(1 - z_{ij}) \quad \forall i \\ & -\mathbf{y}_{ij} + \mathbf{x}_i \boldsymbol{\beta}_j \leq \epsilon z_{ij} + M(1 - z_{ij}) \quad \forall i \\ & \sum_{i=1}^{N_s} z_{ij} \geq N_s(1 - \alpha) \end{aligned}$$

## References

- Yitong Liu, Zhengshuo Li, and Junbo Zhao. “Robust Data-Driven Linear Power Flow Model With Probability Constrained Worst-Case Errors”. In: *IEEE Transactions on Power Systems* 37.5 (2022), pp. 4113–4116. DOI: [10.1109/TPWRS.2022.3189543](https://doi.org/10.1109/TPWRS.2022.3189543)
- Wei Wei. “Tutorials on Advanced Optimization Methods”. In: *arXiv preprint arXiv:2007.13545* (2020)
- Zhentong Shao et al. “A linear AC unit commitment formulation: An application of data-driven linear power flow model”. In: *International Journal of Electrical Power & Energy Systems* 145 (2023), p. 108673



## 4.8 Physical-model-informed Family

This class of functions covers the following algorithms:

- DCPF (DC)
- DCPF with ordinary least squares (DC\_LS)
- Original DLPF (DLPF)
- DLPF with a data-driven correction (DLPF\_C)
- DC-based PTDF to compute branch flow (PTDF)
- First-order Taylor approximation (TAY)

These functions share the common parameter listed in Table 4.36.

Table 4.36: Table of common parameters to all the physical-model-informed linearization methods.

Parameter	Format	Default	Description
<code>opt.idx</code>	function/struct	<code>func_define_idx</code>	A struct (or function calling a struct) that contains the column indices of the from- and to-buses in a MATPOWER-like <code>mpc.branch</code> matrix. By default, it uses the built-in index in MATPOWER.

### 4.8.1 DCPF (DC)

#### Tips

- DCPF assumes that voltage magnitude  $V_m$  is 1 p.u. at all buses, branch resistances are negligible, and voltage angle differences between buses  $i$  and  $j$  are small enough that  $\sin(\theta_i - \theta_j) \approx \theta_i - \theta_j$  for all buses  $i$  and  $j$ .
- This method fixes the predictor as  $P$  and the response as  $V_a$  in `variable.predictor` and `variable.response` respectively. Changing the inputs to these function arguments will have no effect on the output. The remaining predictions are derived from the assumptions of DCPF listed above. Consequently, the branch flows  $PF = -PT$ , and  $QF = QT = 0$ .
- The input data to DC must be unnormalized.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'DC');
```

```
>> model = func_algorithm_DC(data);
```

#### References

Ray D Zimmerman and Carlos E Murillo-Sánchez. “Matpower 6.0 user’s manual”. In: *Power Systems Engineering Research Center* 9 (2016)

### 4.8.2 DCPF with Ordinary Least Squares (DC\_LS)

#### Tips

- DC\_LS assumes that voltage magnitude  $V_m$  is 1 p.u. at all buses, branch resistances are negligible, and voltage angle differences between buses  $i$  and  $j$  are small enough that  $\sin(\theta_i - \theta_j) \approx \theta_i - \theta_j$  for all buses  $i$  and  $j$ .
- This method fixes the predictor as  $P$  and the response as  $V_a$  in `variable.predictor` and `variable.response` respectively. Changing the inputs to these function arguments will have no effect on the output. The remaining predictions are derived from the assumptions, identical to those of DCPF, listed above. Consequently, the branch flows  $PF = -PT$ , and  $QF = QT = 0$ .
- The input data to DC\_LS must be unnormalized.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'DC_LS');
```

```
>> model = func_algorithm_DC_LS(data);
```

#### More About

This method uses the ordinary least squares approach (as in Section 4.2.1) to get the solution

$$\hat{\beta} = \left( \tilde{X}^\top \tilde{X} \right)^{-1} \tilde{X}^\top Y$$

where  $\tilde{X} = XB^{-1}$ .  $X$  comprises the real power of PV and P buses, while  $B$  refers to the susceptance matrix of PV and PQ buses. Accordingly, predictions are made as

$$Y_{\text{pred}} = X_{\text{test}} B^{-1} \hat{\beta}$$

#### References

Xingpeng Li and Kory Hedman. “Data driven linearized ac power flow model with regression analysis”. In: *arXiv preprint arXiv:1811.09727* (2018)

### 4.8.3 Decoupled Linearized Power Flow (DLPF)

#### Tips

- This method computes  $\beta$  to map the known values of  $P$ ,  $Q$ ,  $V_m$ , and  $V_a$  to the unknown values of  $V_m$  and  $V_a$ , together with  $PF$  and  $PT$ . Changing the inputs to the function arguments of `variable.predictor` and `variable.response` will therefore have no effect on the output. Technically,  $QF$  and  $QT$  can be predicted as well, but this is not done in the current version of DLPF.
- The input data to DLPF must be unnormalized.

- The absolute values of predicted voltage angles are significantly different from those of ACPF when the slack bus angle is non-zero. However, the predicted voltage magnitudes and active power line flows exhibit much high accuracy, suggesting that the prediction of angle differences is more important than the prediction of the absolute values of angles. Thus, the error of  $V_a$  and its analysis can be ignored with this method.

### Examples

```
>> model = daline.fit(data, 'method.name', 'DLPF');
```

```
>> model = func_algorithm_DLPF(data);
```

### References

Jingwei Yang et al. “A state-independent linear power flow model with accurate estimation of voltage magnitude”. In: *IEEE Transactions on Power Systems* 32.5 (2016), pp. 3607–3617

#### 4.8.4 DLPF with a Data-driven Correction (DLPF\_C)

##### Tips

- This method computes  $\beta$  to map the known values of  $P$ ,  $Q$ ,  $V_m$ , and  $V_a$  to the unknown values of  $V_m$  and  $V_a$ , together with  $PF$  and  $PT$ . Changing the inputs to the function arguments of `variable.predictor` and `variable.response` will therefore have no effect on the output. Technically,  $QF$  and  $QT$  can be predicted as well, but this is not done in the current version of `DLPF_C`.
- The input data to `DLPF_C` must be unnormalized.
- The absolute values of predicted voltage angles are significantly different from those of ACPF when the slack bus angle is non-zero. However, the predicted voltage magnitudes and active power line flows exhibit much higher accuracy, suggesting that the prediction of angle differences is more important than the prediction of the absolute values of angles. Thus, the error of  $V_a$  and its analysis can be ignored with this method.

### Examples

```
>> model = daline.fit(data, 'method.name', 'DLPF_C');
```

```
>> model = func_algorithm_DLPF_C(data);
```

### More About

This method takes the coefficients  $\beta_{DLPF}$  from Section 4.8.3 and corrects the results. The residuals of DLPF from the training data set are first calculated as

$$\Delta Y = Y - X\beta_{DLPF}$$

$\Delta\mathbf{Y}$  is subsequently used in QR decomposition with  $\mathbf{X}$  (as in Section 4.9.1) to get  $\Delta\boldsymbol{\beta}$ . The corrected coefficients for the linear model are then

$$\hat{\boldsymbol{\beta}} = \boldsymbol{\beta}_{\text{DLPF}} + \Delta\boldsymbol{\beta}$$

## References

Mengshuo Jia et al. “Frequency-Control-Aware Probabilistic Load Flow: An Analytical Method”. In: *IEEE Transactions on Power Systems* 38.6 (2023), pp. 5170–5187

## 4.8.5 Power Transfer Distribution Factor (PTDF)

### Tips

- This method takes the classic DC-based PTDL approach, using MATPOWER’s `makePTDFP` function. Thus, it fixes the predictor as `P` and the response as `PF` in `variable.predictor` and `variable.response` respectively. Changing the inputs to these function arguments will have no effect on the output.
- The results of PTDF should be identical to the active power branch flows of DC.

### Examples

```
>> model = daline.fit(data, 'method.name', 'PTDF');
```

```
>> model = func_algorithm_PTDF(data);
```

## References

Ray D Zimmerman and Carlos E Murillo-Sánchez. “Matpower 6.0 user’s manual”. In: *Power Systems Engineering Research Center* 9 (2016)

## 4.8.6 First-order Taylor Approximation (TAY)

### Additional inputs

Table 4.37: Table of parameters specific to first-order Taylor approximation.

Parameter	Format	Default	Description
TAY.point0	character/integer	'end'	Index of the expansion point in the training dataset; should be less than or equal to <code>num.trainSample</code> , e.g., TAY.point0 = 200 when <code>num.trainSample</code> is 300. If its argument is 'end', the method uses the last sample in the training dataset as the operating point. When the data are time series data, using the last sample as the expansion point indicates that this method is a warm-start Taylor approximation.

### Tips

- This method fixes the predictors as P, Q and the responses as Vm, Va in `variable.predictor` and `variable.response` respectively. Changing the inputs to these function arguments will have no effect on the output.

### Examples

```
>> model = daline.fit(data, 'method.name', 'TAY');
```

```
>> model = daline.fit(data, 'method.name', 'TAY', 'TAY.point0', 1);
```

```
>> opt = daline.setopt('method.name', 'TAY', 'TAY.point0', 1);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_TAY(data, 'TAY.point0', 1);
```

```
>> opt = daline.setopt('TAY.point0', 1);
>> model = func_algorithm_TAY(data, opt);
```

### More About

This method relies on computation of the Jacobian matrix

$$\mathbf{J} = \frac{d\mathbf{x}}{d\mathbf{y}}$$

where  $\mathbf{x}$  refers to the active and reactive power injections, and  $\mathbf{y}$  to voltage magnitudes and angles, of a single observation. At around operating point 0,

$$\mathbf{J}_0 = \left. \frac{d\mathbf{x}}{d\mathbf{y}} \right|_0 \approx \frac{\mathbf{x} - \mathbf{x}_0}{\mathbf{y} - \mathbf{y}_0}$$

A first-order Taylor approximation around  $\mathbf{y}$  can therefore be derived as

$$\mathbf{y} \approx \mathbf{y}_0 + \mathbf{J}_0^{-1} \mathbf{x} - \mathbf{J}_0^{-1} \mathbf{x}_0$$

which can also be represented as

$$\mathbf{y}^\top = \begin{bmatrix} 1 & \mathbf{x}^\top \end{bmatrix} \begin{bmatrix} \mathbf{y}_0^\top - \mathbf{x}_0^\top \mathbf{J}_0^{-1\top} \\ \mathbf{J}_0^{-1\top} \end{bmatrix}$$

for each data point  $\mathbf{x}$ .

Thus,

$$\hat{\boldsymbol{\beta}} = \begin{bmatrix} \mathbf{y}_0^\top - \mathbf{x}_0^\top \mathbf{J}_0^{-1\top} \\ \mathbf{J}_0^{-1\top} \end{bmatrix}$$

## References

Xingpeng Li and Kory Hedman. “Data driven linearized ac power flow model with regression analysis”. In: *arXiv preprint arXiv:1811.09727* (2018)

## 4.9 Direct Solution Family

This class of functions covers the following algorithms:

- Direct QR decomposition (QR)
- Direct left division (LD)
- Direct generalized inverse (PIN)
- Direct singular value decomposition (SVD)
- Direct complete orthogonal decomposition (COD)
- Direct principal component analysis (PCA)

### 4.9.1 Direct QR Decomposition (QR)

#### Examples

```
>> model = daline.fit(data, 'method.name', 'QR');
```

```
>> model = daline.fit(data, 'method.name', 'QR', 'variable.predictor', {'P'}, 'variable.  
response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'QR', 'variable.predictor', {'P', 'Q'}, 'variable.  
response', {'PF'});  
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_QR(data, 'variable.predictor', {'P', 'Q'}, 'variable.  
response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});  
>> model = func_algorithm_QR(data, opt);
```

#### More About

This method factorizes the predictor matrix  $\mathbf{X} \in \mathbb{R}^{N_s \times N_x}$  using the QR decomposition  $\mathbf{X} = \mathbf{Q}\mathbf{R}$ , where  $\mathbf{Q}$  is an  $N_s$ -by- $N_s$  orthogonal matrix and  $\mathbf{R}$  is an  $N_s$ -by- $N_x$  upper-triangular matrix.

Subsequently, back-substitution is used in

$$\mathbf{R}\hat{\boldsymbol{\beta}} = \mathbf{Q}^\top \mathbf{Y}$$

to find  $\hat{\boldsymbol{\beta}}$ .

## 4.9.2 Direct Left Division (LD)

### Examples

```
>> model = daline.fit(data, 'method.name', 'LD');
```

```
>> model = daline.fit(data, 'method.name', 'LD', 'variable.predictor', {'P'}, 'variable.  
.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'LD', 'variable.predictor', {'P', 'Q'}, 'variable.  
.response', {'PF'});  
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_LD(data, 'variable.predictor', {'P', 'Q'}, 'variable.  
.response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});  
>> model = func_algorithm_LD(data, opt);
```

### More About

The  $\backslash$  operator in MATLAB represents matrix left division, such that  $X = A \backslash B$  solves the system of linear equations  $A * X = B$  for  $X$ . In this method,  $\backslash$  is used to compute  $\beta$  in

$$\hat{\beta} = X^{-1}Y$$

## 4.9.3 Direct Generalized Inverse (PIN)

### Tips

- The  $\hat{\beta}$  found by PIN and LS\_PIN generally do not match.
- Preliminary empirical testing suggests that PIN generally achieves a smaller mean relative error compared to LS\_PIN.

### Examples

```
>> model = daline.fit(data, 'method.name', 'PIN');
```

```
>> model = daline.fit(data, 'method.name', 'PIN', 'variable.predictor', {'P'}, '  
.variable.response', {'PF', 'Vm'});
```



```
>> opt = daline.setopt('method.name', 'PIN', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PIN(data, 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_PIN(data, opt);
```

## More About

The generalized inverse (a.k.a. the Moore-Penrose pseudoinverse) is used in this method to find the minimum  $L^2$  norm solution to a system of linear equations with infinite solutions. Specifically, the generalized inverse is implemented in

$$\hat{\beta} = \mathbf{X}^{-1}\mathbf{Y}$$

## 4.9.4 Direct Singular Value Decomposition (SVD)

### Tips

- The  $\hat{\beta}$  found by SVD and LS\_SVD generally do not match. SVD tends to return a sparser set of  $\hat{\beta}$  (i.e., more linear coefficients are set to zero).
- Preliminary empirical testing suggests that SVD generally achieves a smaller mean relative error compared to LS\_SVD. However, the difference narrows with larger bus systems and data set sizes.

### Examples

```
>> model = daline.fit(data, 'method.name', 'SVD');
```

```
>> model = daline.fit(data, 'method.name', 'SVD', 'variable.predictor', {'P'}, 'variable.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'SVD', 'variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_SVD(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_SVD(data, opt);
```

## More About

This method factorizes the predictor matrix  $\mathbf{X} \in \mathbb{R}^{N_s \times N_x}$  using the SVD  $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ , where  $\mathbf{U}$  is an  $N_s$ -by- $N_s$  orthogonal matrix,  $\mathbf{\Sigma}$  is an  $N_s$ -by- $N_x$  matrix with singular values on its diagonal, and  $\mathbf{V}^\top$  is an  $N_x$ -by- $N_x$  orthogonal matrix. Its pseudoinverse is then implemented to find  $\hat{\beta}$  in

$$\hat{\beta} = \mathbf{X}^{-1}\mathbf{Y} = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top)^{-1}\mathbf{Y}$$

## 4.9.5 Direct Complete Orthogonal Decomposition (COD)

### Tips

- The  $\hat{\beta}$  found by COD and LS\_COD generally do not match. COD tends to return a sparser set of  $\hat{\beta}$  (i.e., more linear coefficients are set to zero).
- Preliminary empirical testing suggests that COD generally achieves a smaller mean relative error compared to LS\_COD. However, the difference narrows with larger bus systems and data set sizes.

### Examples

```
>> model = daline.fit(data, 'method.name', 'COD');
```

```
>> model = daline.fit(data, 'method.name', 'COD', 'variable.predictor', {'P'}, '
    variable.response', {'PF', 'Vm'});
```

```
>> opt = daline.setopt('method.name', 'COD', 'variable.predictor', {'P', 'Q'}, 'variable
    .response', {'PF'});
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_COD(data, 'variable.predictor', {'P', 'Q'}, 'variable.
    response', {'PF', 'QF'});
```

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF'});
>> model = func_algorithm_COD(data, opt);
```

### More About

This method factorizes the predictor matrix  $\mathbf{X} \in \mathbb{R}^{N_s \times N_x}$  using complete orthogonal decomposition to get

$$\mathbf{X} = \tilde{\mathbf{Q}} \begin{bmatrix} \tilde{\mathbf{R}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{Z}^\top$$

where  $\tilde{\mathbf{Q}} \in \mathbb{R}^{N_s \times N_s}$  and  $\mathbf{Z} \in \mathbb{R}^{N_s \times N_s}$  are both orthogonal matrices, and  $\tilde{\mathbf{R}} \in \mathbb{R}^{N_{sr} \times N_{sr}}$  is an upper-triangular matrix. Its pseudoinverse is then implemented to find  $\hat{\beta}$  in

$$\hat{\beta} = \mathbf{X}^{-1} \mathbf{Y} = \mathbf{Z} \begin{bmatrix} \tilde{\mathbf{R}}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \tilde{\mathbf{Q}}^\top \mathbf{Y}$$

### 4.9.6 Direct Principal Component Analysis (PCA)

#### Additional inputs

Refer to Table 4.5 for additional inputs of this method.

#### Tips

- The  $\hat{\beta}$  found by PCA and LS\_PCA generally do not match.
- Preliminary empirical testing suggests that PCA and LS\_PCA generally achieve comparable relative errors.
- See also the tips in Section 4.2.5.

#### Examples

```
>> model = daline.fit(data, 'method.name', 'PCA');
```

```
>> model = daline.fit(data, 'method.name', 'PCA', 'variable.predictor', {'P'}, 'variable.response', {'PF', 'Vm'}, 'PCA.PerComponent', [30:10:90], 'PCA.numFold', 5);
```

```
>> opt = daline.setopt('method.name', 'PCA', 'variable.predictor', {'P'}, 'variable.response', {'PF', 'Vm'}, 'PCA.PerComponent', [30:10:90], 'PCA.numFold', 5);
>> model = daline.fit(data, opt);
```

```
>> model = func_algorithm_PCA(data, 'variable.predictor', {'P'}, 'variable.response',
    {'PF', 'Vm'}, 'PCA.PerComponent', [30:10:90], 'PCA.numFold', 5);
```

```
>> opt = daline.setopt('variable.predictor', {'P'}, 'variable.response', {'PF', 'Vm'}, '
    PCA.PerComponent', [30:10:90], 'PCA.numFold', 5);
>> model = func_algorithm_PCA(data, opt);
```

### More About

The underlying idea behind this method is to project the predictor matrix  $\mathbf{X}$  onto a new orthonormal basis; i.e., convert the features in the data into uncorrelated features (“principal components”) before training. The covariance of  $\mathbf{X}$  undergoes eigendecomposition as

$$\text{Cov}(\mathbf{X}) = \mathbf{D}\mathbf{\Lambda}\mathbf{D}^\top$$

where  $\mathbf{D} \in \mathbb{R}^{N_x \times N_x}$  consists of the eigenvectors of  $\text{Cov}(\mathbf{X})$ , and  $\mathbf{\Lambda}$  is a diagonal matrix of the corresponding eigenvalues. The system of linear equations  $\mathbf{Y} = \mathbf{X}\mathbf{D}\hat{\boldsymbol{\beta}}^{PCA}$  is directly solved via

$$\hat{\boldsymbol{\beta}}^{PCA} = (\mathbf{X}\mathbf{D})^{-1}\mathbf{Y} \quad (4.6)$$

such that the linear coefficients of the original predictor data are given as

$$\hat{\boldsymbol{\beta}} = \mathbf{D}\hat{\boldsymbol{\beta}}^{PCA}$$

## Chapter 5

# Performance Evaluation and Visualization

DALINE supports multiple visualization modules that can generate different graphical outputs. These outputs focus on linearization accuracy and computational efficiency. Specifically, the package includes: `daline.rank` (`daline.plot`) and `daline.time`. Specifically,

- `daline.rank` executes and ranks built-in linearization methods in terms of their accuracy, offering various visualization options for comprehensive comparisons using different types, themes, styles, and patterns. Once the execution of methods is completed and the linearization results are acquired, `daline.plot` can substitute `daline.rank` for visualization purposes.
- `daline.time` executes and compares built-in linearization methods in terms of their computational efficiency and scalability. It also offers various visualization options.

### 5.1 Accuracy

The wrapper `daline.rank` is primarily designed to execute multiple linearization methods within a given test system, producing various `models` as outputs. This wrapper then evaluates and ranks the relative linearization errors of these methods. The ranking is based on the mean value of the relative errors for the selected response(s), such as the active branch flow PF. Additionally, `daline.rank` identifies and returns the names of the methods that failed to generate a valid linear model (e.g., those with coefficients containing NaN<sup>1</sup>). Ultimately, `daline.rank` plots and ranks the relative error distributions of the selected response(s) for all the chosen methods, according to user-defined settings for themes, styles, patterns, etc. `daline.rank` offers two types of error distributions: the moment-based distribution characterized by the mean and maximum relative errors (plus the minimum relative error if a certain theme is chosen), and the probability-based distribution fitted by Gaussian mixture models. Certainly, if only one method is selected, the outputted `model` and ranking reflect the results of that single method.

#### Inputs

First, the two essential inputs for `daline.rank` are subject to specific format requirements. The `data` input must fit the standardized data format defined by DALINE, as detailed in Section 3.2.1. If `data` is generated by DALINE, it can be seamlessly utilized in `daline.rank`. Furthermore, the list of methods

---

<sup>1</sup>Failures can be attributed to various reasons; see our previous work in [3] for a detailed failure analysis over exhaustive simulations.

should be specified in a cell array format, e.g., `methodList = {'LS'; 'LS_SVD'; 'LS_COD'}`, where each entry corresponds to the name of a built-in linearization method.

Additionally, `daline.rank` accommodates a wide range of parameters due to its support for model training and testing across multiple methods. All parameters applicable to these methods can be integrated into `daline.rank`. Users can refer to Chapter 4 for details on all specific parameters related to the built-in linearization methods.

Last but not least, `daline.rank` supports numerous options for visualization, as listed below in Table 5.1.

Table 5.1: Table of parameters specific to `daline.rank`

Parameter	Format	Default	Description
<code>PLOT.switch</code>	integer	1	Set to 1 to produce figures; set to 0 otherwise.
<code>PLOT.response</code>	list	(empty)	Specifies the responses to plot, such as {'Vm', 'Va'}, {'PF', 'QF'}. If no valid responses are found (e.g., the error of a response is missing or the error contains 'NaN'), the method is considered a failure. If <code>PLOT.response</code> is not specified, the errors of all the responses will be plotted.
<code>PLOT.type</code>	character	'moment'	Choose 'probability' to plot the probability distribution of error; select 'moment' to plot the moment distribution (min, max, mean) of error.
<code>PLOT.style</code>	character	'dark'	Select 'dark' to plot the error in a dark environment or 'light' for a light environment. Note: <code>PLOT.style</code> is only valid when <code>PLOT.theme = 'commercial'</code> .
<code>PLOT.theme</code>	character	'commercial'	For moment error distribution only: choose 'commercial' for a Big Tech company theme; select 'academic' for a standard academic paper theme.
<code>PLOT.pattern</code>	character	'indivi'	For moment error distribution only: choose 'indivi' to paint the error individually for each response or 'joint' to combine the errors of all given responses into one figure.
<code>PLOT.startAlpha</code>	float	0.95	For probability error distribution only: the transparency level of the first distribution (the most accurate one). Here, value 1 refers to no transparency.

Continued on next page

Table 5.1 – continued from previous page

Parameter	Format	Default	Description
<code>PLOT.endAlpha</code>	float	0.4	For probability error distribution only: the transparency level of the last distribution (the most inaccurate one). Here, value 1 refers to no transparency.
<code>PLOT.disPoints</code>	integer	1000	For probability error distribution only: specifies the number of points used to draw the probability distribution. The larger the number, the smoother the distribution.
<code>PLOT.logShift</code>	float	1e-6	For probability error distribution only: add a small constant to the errors to handle the log transformation for zero error values; this is used for painting only, and this constant will not be added to the quantified result.
<code>PLOT.numComponent</code>	integer	3	For probability error distribution only: specifies the number of components in the Gaussian mixture model used to fit the error probability distribution. When the number of testing data points is small, consider using fewer components.
<code>PLOT.norm</code>	binary	0	For probability error distribution only: 0 refers to showing the original distribution; 1 refers to showing the normalized distribution (this is suggested when the differences among different distributions are huge and the distributions cannot be visualized effectively in the same figure).
<code>PLOT.titleHeight</code>	float	0.5	The larger the <code>titleHeight</code> , the closer the title is to the figure.
<code>PLOT.origin</code>	integer	15	Sets the origin of the log-scaled x-axis to 1e-15 if <code>origin</code> is set to 15.
<code>PLOT.caseName</code>	character	(empty)	Specifies a case name to display in the title of the figure, if provided, e.g., 'IEEE_118_bus_system'; it remains empty by default.

Continued on next page

Table 5.1 – continued from previous page

Parameter	Format	Default	Description
PLOT.print	integer	0	For 'commercial' and 'indivi' styles, set to 1 to print the figure to a PDF and save it in the current folder; set to 0 to not print. The filename of the PDF will be a combination of <code>PLOT.caseName</code> and the response drawn. If <code>PLOT.caseName</code> has not been specified, the filename of the PDF will be “file-name”, which can be overwritten if the print command is executed more than once.

## Examples

```
>> data = daline.data('case.name', 'case118', 'data.baseType', 'TimeSeriesRand', 'load.
    upperRangeTime', 1.05, 'load.lowerRangeTime', 0.95, 'load.distribution', 'normal',
    'voltage.distribution', 'normal', 'voltage.varyIndicator', 0, 'data.parallel', 1,
    'data.fixRand', 1);
>> methodList = {'PLS_CLS'; 'RR_KPC'; 'LS_PCA'; 'LS_LIFX'; 'PLS_RECW'; 'DLPF_C'; 'RR_WEI'
    ; 'RR_VCS'; 'PTDF'; 'DC'; 'DLPF'; 'DC_LS'; 'TAY'};
>> opt = daline.setopt('variable.response', {'PF', 'Vm'}, 'PLOT.switch', 1, 'PLOT.
    response', {'PF'}, 'PLOT.type', 'probability', 'PLOT.style', 'light', 'PLS.
    clusNumInterval', 8, 'RR.clusNumInterval', 12, 'RR.etaInterval', 1000, 'PCA.
    PerComponent', 90, 'PLS.omega', 0.8, 'RR.tauInterval', 0.4, 'RR.lambdaInterval', 1
    e-10);
>> models = daline.rank(data, methodList, opt);
% See the output figure below.
```



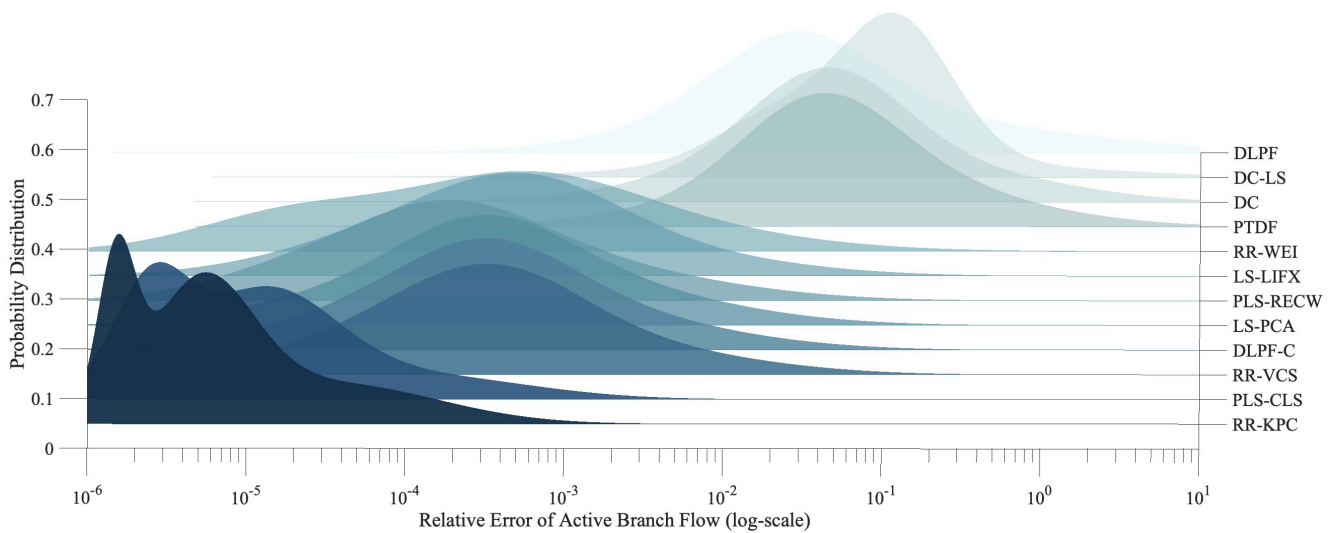


Figure 5.1: Probability distributions for relative errors of the given methods w.r.t. active branch flows in a light style

```
>> data = daline.data('case.name', 'case9');
>> methodList = {'LS'; 'PLS_SIMRX'; 'LS_COD'; 'LS_LIFX'; 'LS_CLS'; 'RR'; 'RR_KPC'; '
    RR_WEI'; 'LS_PIN'; 'LS_PCA'; 'PLS_NIP'; 'PLS_CLS'; 'TAY'; 'DC'; 'PTDF'; 'DLPF'; 'DLPF_C'
    ; 'DC_LS'};
>> daline.rank(data, methodList, 'variable.response', {'Vm', 'PF'}, 'PLOT.response', {'
    Vm', 'PF'});
% See the output figure below. Note that two figures were generated, one for 'Vm'
    and one for 'PF'. Below, only the figure for 'Vm' is shown.
% (The data and methodList will be used for the following three examples.)
```

Relative Error of Vm (log-scale)		
RR_KPC	mean: $7.8 \times 10^{-7}$	max: $2.1 \times 10^{-5}$
LS_LIFX	mean: $4.0 \times 10^{-6}$	max: $3.6 \times 10^{-5}$
PLS_CLS	mean: $4.9 \times 10^{-6}$	max: $5.9 \times 10^{-5}$
RR_WEI	mean: $1.5 \times 10^{-5}$	max: $1.2 \times 10^{-4}$
LS_PIN	mean: $1.6 \times 10^{-5}$	max: $1.2 \times 10^{-4}$
PLS_NIP	mean: $1.6 \times 10^{-5}$	max: $1.2 \times 10^{-4}$
DLPF_C	mean: $1.6 \times 10^{-5}$	max: $1.2 \times 10^{-4}$
LS_COD	mean: $1.6 \times 10^{-5}$	max: $1.2 \times 10^{-4}$
LS_PCA	mean: $1.6 \times 10^{-5}$	max: $1.2 \times 10^{-4}$
RR	mean: $1.6 \times 10^{-5}$	max: $1.2 \times 10^{-4}$
TAY	mean: $2.3 \times 10^{-5}$	max: $1.8 \times 10^{-4}$
PLS_SIMRX	mean: $7.8 \times 10^{-5}$	max: $3.0 \times 10^{-4}$
DLPF	mean: $8.0 \times 10^{-3}$	max: $1.3 \times 10^{-2}$
DC_LS	mean: $1.2 \times 10^{-2}$	max: $3.8 \times 10^{-2}$
DC	mean: $1.2 \times 10^{-2}$	max: $3.8 \times 10^{-2}$
PTDF		failure
LS_CLS		failure
LS		failure

Figure 5.2: Moment distributions for relative errors of the given methods w.r.t. voltage magnitudes in a dark, commercial theme.

```
>> daline.rank(data, methodList, 'variable.response', {'Vm', 'PF'}, 'variable.liftType', 'gauss', 'TAY.point0 ', 'end', 'PLOT.response', {'PF'}, 'PLOT.style', 'light');
% Only the parameters of methods 'LS_LIFX' and 'TAY' have been adjusted here, but
  users can specify more. See the output figure below.
```

Relative Error of PF (log-scale)		
RR_KPC	mean: $1.7 \times 10^{-6}$	max: $8.6 \times 10^{-5}$
LS_LIFX	mean: $9.4 \times 10^{-6}$	max: $1.5 \times 10^{-4}$
PLS_CLS	mean: $1.1 \times 10^{-5}$	max: $2.4 \times 10^{-4}$
RR_WEI	mean: $3.6 \times 10^{-5}$	max: $4.9 \times 10^{-4}$
LS_PCA	mean: $3.7 \times 10^{-5}$	max: $5.1 \times 10^{-4}$
DLPF_C	mean: $3.7 \times 10^{-5}$	max: $5.1 \times 10^{-4}$
PLS_NIP	mean: $3.7 \times 10^{-5}$	max: $5.1 \times 10^{-4}$
LS_PIN	mean: $3.7 \times 10^{-5}$	max: $5.1 \times 10^{-4}$
LS_COD	mean: $3.7 \times 10^{-5}$	max: $5.1 \times 10^{-4}$
RR	mean: $3.7 \times 10^{-5}$	max: $5.1 \times 10^{-4}$
PLS_SIMRX	mean: $2.2 \times 10^{-3}$	max: $9.0 \times 10^{-3}$
DC_LS	mean: $1.7 \times 10^{-2}$	max: $6.6 \times 10^{-2}$
PTDF	mean: $2.3 \times 10^{-2}$	max: $6.7 \times 10^{-2}$
DC	mean: $2.3 \times 10^{-2}$	max: $6.7 \times 10^{-2}$
DLPF	mean: $5.5 \times 10^{-1}$	max: $5.1 \times 10^0$
TAY		failure
LS_CLS		failure
LS		failure

Figure 5.3: Moment distributions for relative errors of the given methods w.r.t. active branch flows in a light, commercial theme.

```
>> [models, failure] = daline.rank(data, methodList, 'PLOT.response', {'Vm', 'PF'}, '
    PLOT.theme', 'academic', 'PLOT.pattern', 'joint');
% In the output, 'models' is a struct; each field contains the training and testing
    results of each given method.
% In the output, 'failure' indicates which method(s) failed when predicting the
    responses in 'PLOT.response'. Note that for the 'joint' pattern, a method is
    considered a failure only if this method failed to predict all the responses
    given in 'PLOT.response'.
% See the output figure below.
```

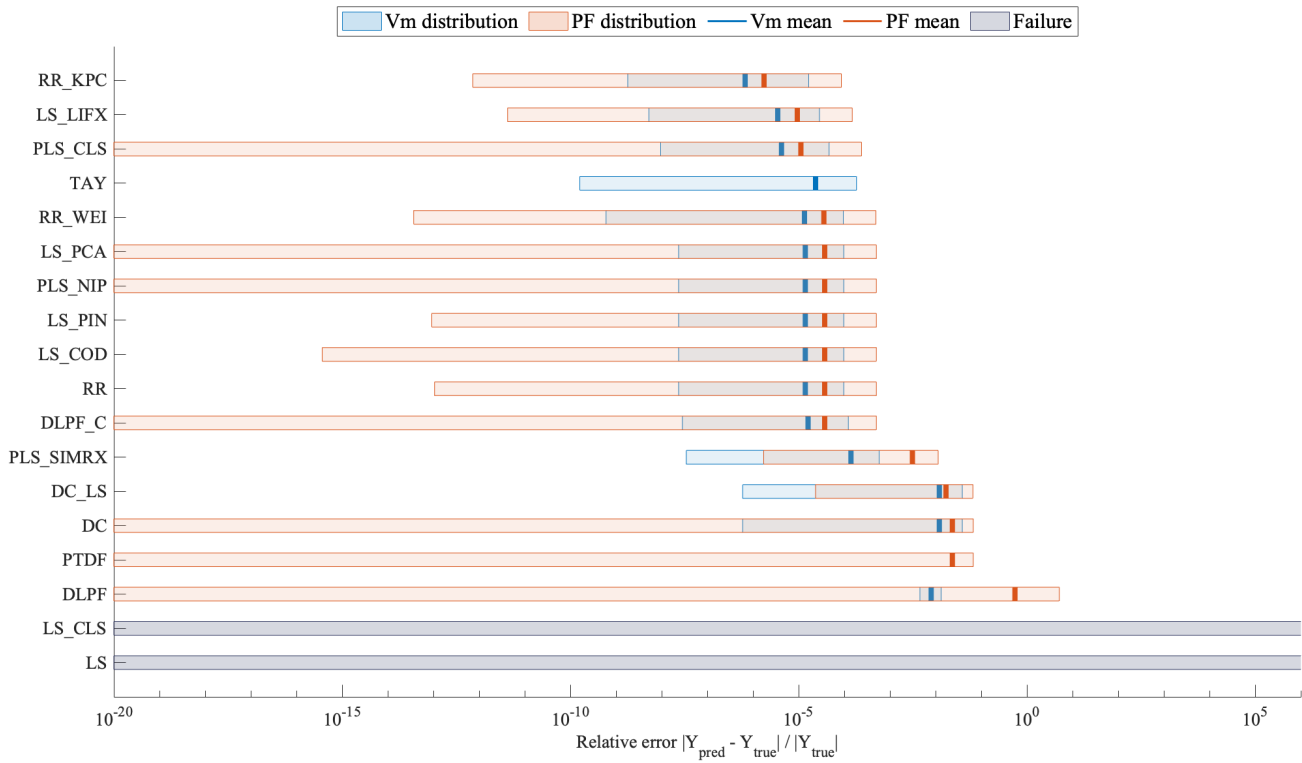


Figure 5.4: Moment distributions for relative errors of the given methods w.r.t. voltage magnitudes in an academic, joint theme.

```
>> opt = daline.setopt('variable.response', {'Vm', 'PF'}, 'PLOT.response', {'Vm', 'PF'},
    'PLOT.theme', 'academic', 'PLOT.pattern', 'indivi');
>> [models, failure] = daline.rank(data, methodList, opt);
```

*% Like the other wrappers, daline.rank also accepts the structured 'opt' formulated by daline.setopt.*

*% In the output, 'models' is a struct; each field contains the training and testing results of each given method.*

*% In the output, the 'failure' indicator identifies which method(s) failed when predicting the responses specified in \texttt{PLOT.response}. For the 'indivi' pattern, a method is considered to have failed if it fails to predict any of the responses in \texttt{PLOT.response} at least once. Additionally, 'failure' also documents which specific responses caused a method to fail.*

*% See the output figures below.*

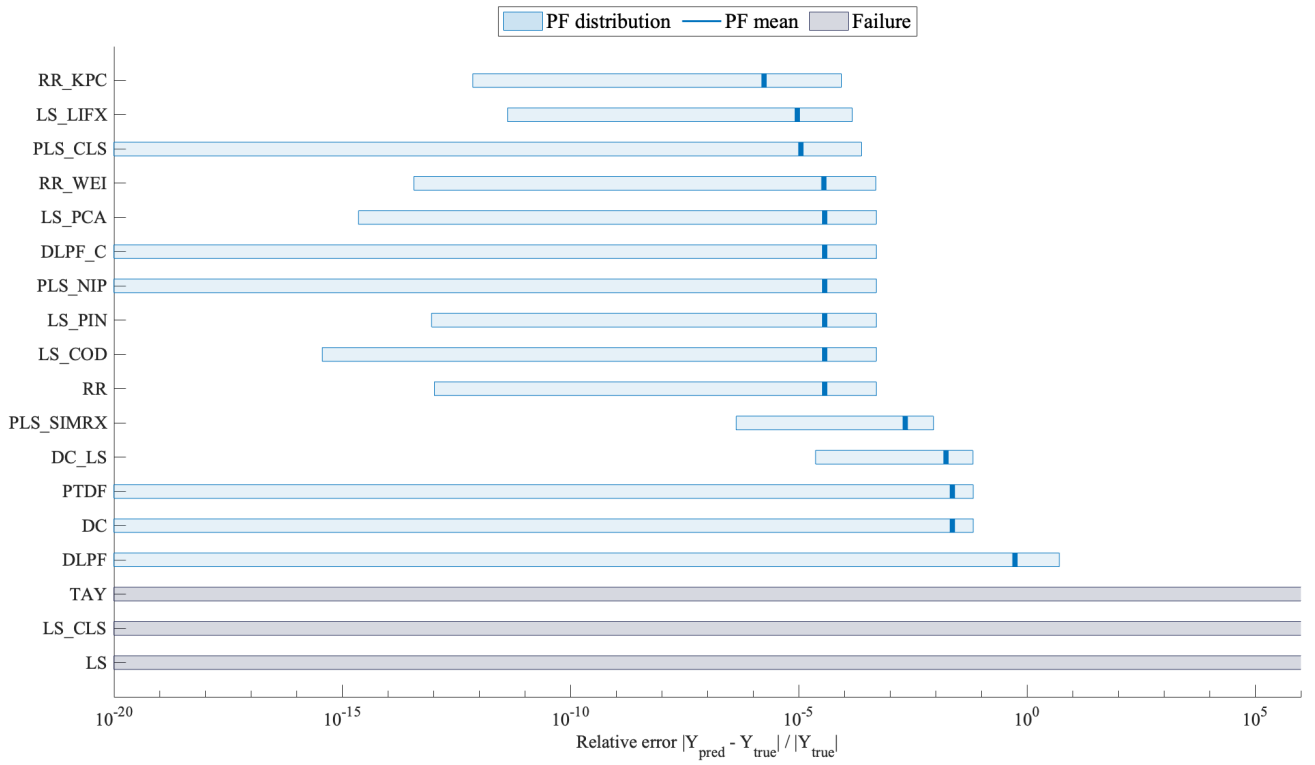


Figure 5.5: Moment distributions for relative errors of the given methods w.r.t. active branch flows in an academic, individual theme.

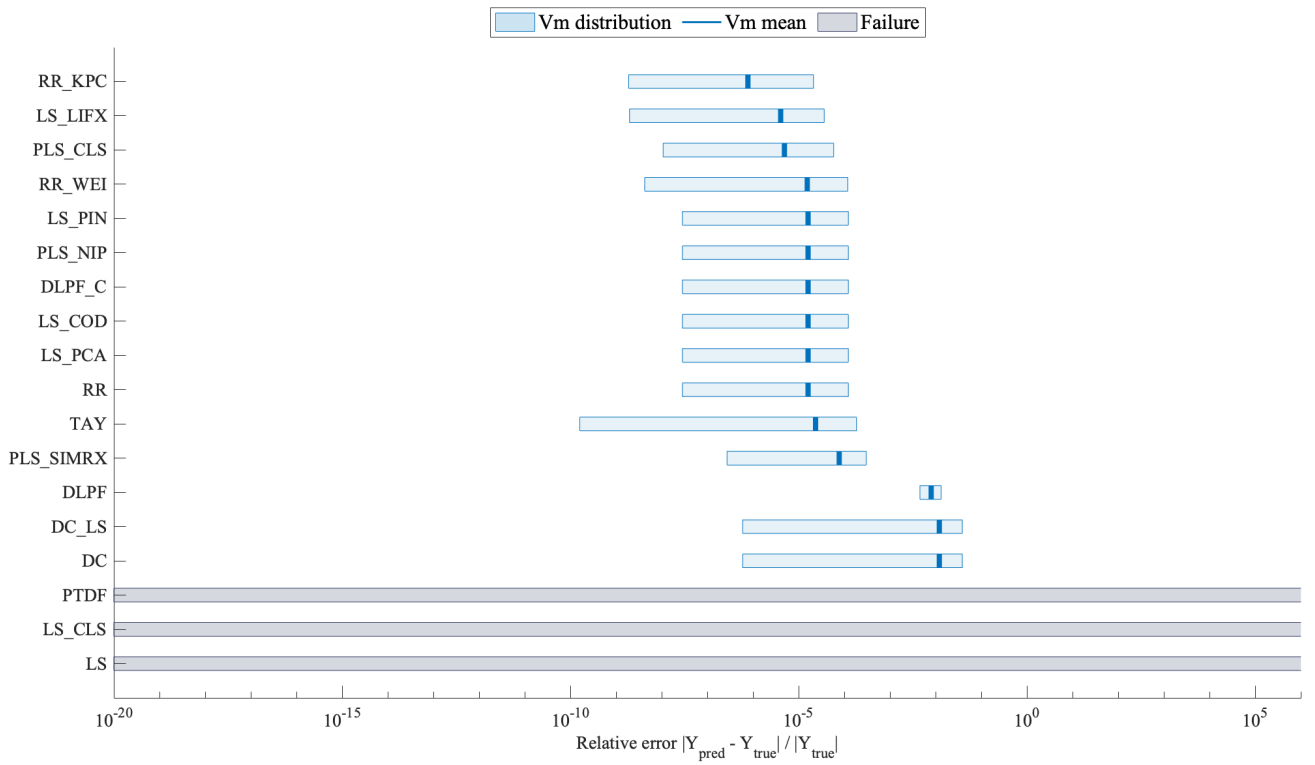


Figure 5.6: Moment distributions for relative errors of the given methods w.r.t. voltage magnitudes in an academic, individual theme.

*% More examples regarding how to adjust the parameters specific to particular methods when comparing them. Users only need to put the name-value pair parameters from the third argument of `daline.rank`. For the parameters of methods, no mandatory orders or mapping constraints are enforced.*

```
>> data = daline.data('case.name', 'case9');
>> daline.rank(data, {'DLPF_C', 'RR', 'PLS_REC'}, 'RR.lambdaInterval', 1e-5, 'RR.
    cvNumFold', 4, 'PLS.recursivePercentage', 40);
```

*% More examples regarding how to adjust the parameters specific to particular methods when comparing them. Users can also use `opt` parameter structure, which can be used as the third argument of `daline.rank`*

```
>> data = daline.data('case.name', 'case9');
>> opt = daline.setopt('RR.lambdaInterval', 1e-5, 'RR.cvNumFold', 4, 'PLS.
    recursivePercentage', 40);
>> daline.rank(data, {'DLPF_C', 'RR', 'PLS_REC'}, opt);
```

*% More examples regarding how to adjust the parameters specific to particular methods when comparing them. Certainly, different methods can always be collected into a set, e.g.,*

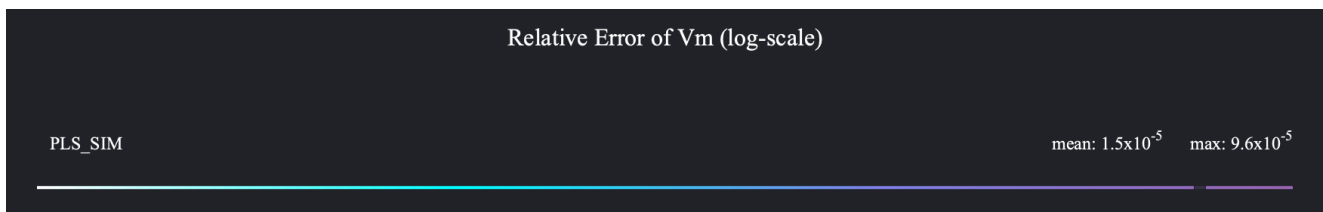
```
>> data = daline.data('case.name', 'case9');
>> method = {'DLPF_C', 'RR', 'PLS_REC'};
>> opt = daline.setopt('RR.lambdaInterval', 1e-5, 'RR.cvNumFold', 4, 'PLS.
    recursivePercentage', 40);
>> daline.rank(data, method, opt);
```

## Tips

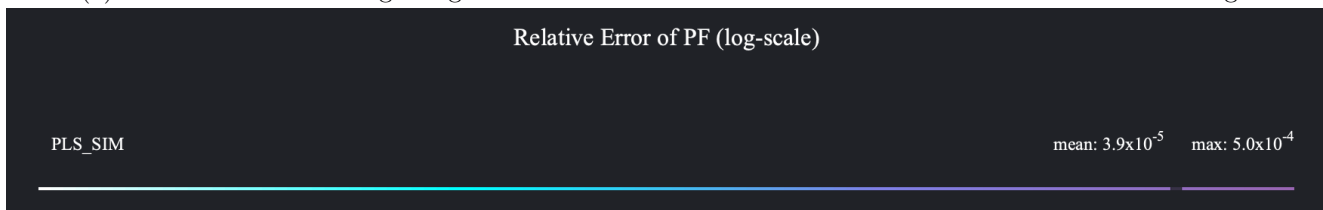
- When users need to compare and rank multiple methods, they may wish to adjust numerous parameters for these methods. Manually typing the name-value pairs of these parameters and entering them into `daline.rank` can be cumbersome and inelegant. Instead, it is advisable for users to modify the file that holds the default parameters, named `func_default_option_category`. After setting these default parameters to meet their requirements, users can then straightforwardly call `daline.rank(data, method)` for comparison and ranking purposes, eliminating the need to manually input a large number of name-value pairs.
- When users opt to plot probability distributions, be aware that these distributions are fitted using a Gaussian mixture model via the expectation-maximization method. This approach may encounter overfitting issues if the number of testing data points is insufficient. To mitigate this, consider using a larger dataset to generate more error data, reducing the number of Gaussian components by adjusting `PLOT.numComponent`, or simply re-running the function.
- If users attempt to plot probability distributions but the resulting figure lacks visible distributions, this is typically due to significant differences in distribution scales, causing many distributions to be compressed and invisible. In such cases, users can utilize normalized probability distributions by setting `PLOT.norm` to 1.

- In certain situations, users may only need to replot or modify the figures without rerunning the entire process. However, re-executing `daline.rank` triggers an unnecessary and time-consuming retraining and retesting of models across multiple methods. To avoid this, the output `models` from `daline.rank` can be directly used as the primary argument in `daline.plot`. The options listed in Table 5.1 apply to `daline.plot` as well.
- Except for taking the output `models` from `daline.rank`, `daline.plot` can also accept the single model output from `daline.fit`. See below for the examples in this regard.

```
>> data = daline.data('case.name', 'case9');
>> model = daline.fit(data, 'method.name', 'PLS_SIM');
>> daline.plot(model);
% Will generate several figures. Two of them are shown below.
% (Note that the following four examples are also based on 'case9' and the method 'PLS_SIM').
```



(a) Relative error of voltage magnitude of PLS\_SIM under MATPOWER `case9` with default settings



(b) Relative error of active branch flow of PLS\_SIM under MATPOWER `case9` with default settings

Figure 5.7: Demonstration of the dark, commercial theme when plotting the errors individually. Note the small dark block on the bar represents the mean error, and the right end of the bar is the maximal error. Also note that the x-axis is at a log scale.

```
>> daline.plot(model, 'PLOT.response', {'Vm'}, 'PLOT.style', 'light');
% One figure of 'Vm' using the light, commercial theme, as shown below.
```

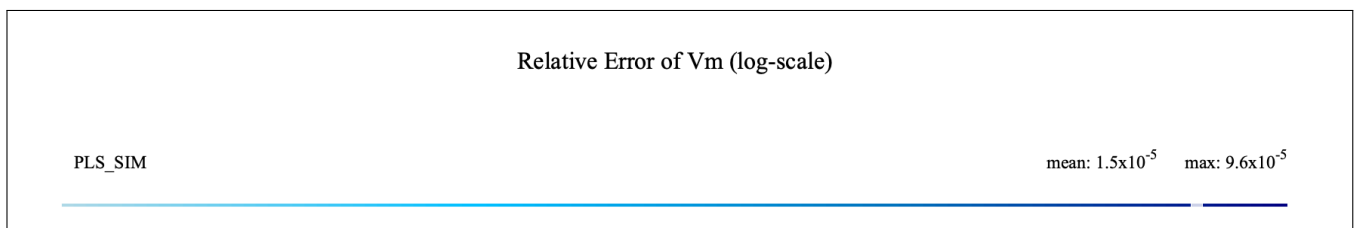


Figure 5.8: Demonstration of the light, commercial theme for the error of the voltage magnitude only. Note the small light block on the bar represents the mean error, and the right end of the bar is the maximal error. Also note that the x-axis is at a log scale.

```
>> daline.plot(model, 'PLOT.response', {'Vm', 'PF'}, 'PLOT.theme', 'academic', 'PLOT.
style', 'light', 'PLOT.pattern', 'joint');
% One figure that jointly shows the errors of 'Vm' and 'PF' using the academic theme,
as shown below.
```

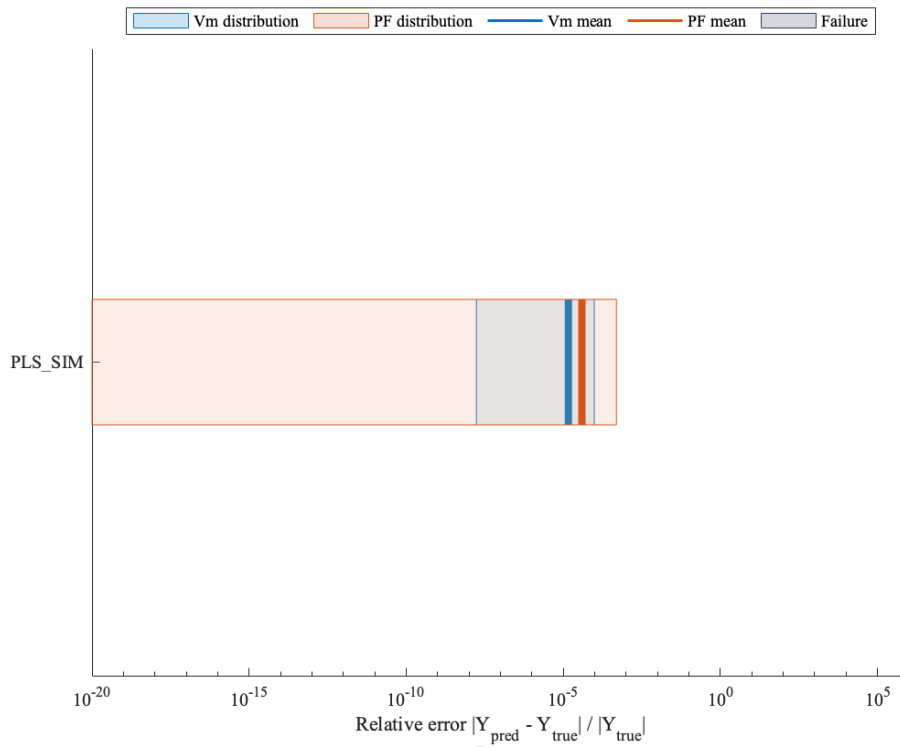


Figure 5.9: Demonstration of the academic, joint theme for the relative errors of voltage magnitude and active branch flow. The blue bar indicates the error distribution of voltage magnitude: the left end is the minimum error, the right end is the maximum error, and the bold blue line within the bar is the mean error. So is the active branch flow.

```
>> failure = daline.plot(model, 'PLOT.response', {'Vm', 'PF', 'Va'}, 'PLOT.switch', 0);
% Generate no figures, but provide the indicator of failure. The indicator
determines if the method 'PLS_SIM' encounters any failures while employing the
linear model it generated to compute 'Vm', 'PF', or 'Va'. Any failure details are
methodically organized in the output labeled 'failure'. If there are no failures
, this output will remain empty.
```

```
>> opt = daline.setopt('PLOT.response', {'Vm', 'PF'}, 'PLOT.theme', 'academic', 'PLOT.
pattern', 'indivi');
>> failure = daline.plot(model, opt);
% Like the other wrappers, daline.plot also accepts the structured 'opt' formulated
by daline.setopt.
```



## 5.2 Visualization of Computational Efficiency

For a specific power system case, `daline.time` can measure and compare the computational times of various built-in methods in terms of their model training and testing. Additionally, for multiple power system cases of varying sizes, `daline.time` can generate computational time evolution curves for these methods, to show the scalability of these approaches in terms of computational burden. The output of `daline.time`, except for the graphical outcomes, is a `timeList`, recording the computational times of the given methods.

### 5.2.1 Computational time rankings of multiple methods (`daline.time`)

#### Inputs

When comparing the computational times of various built-in methods for a test case, the required inputs for `daline.time` include a `data` struct and a list of method names. The `data` input, corresponding to the test case, must conform to the standardized format defined by DALINE, as detailed in Section 3.2.1. If `data` is generated by DALINE, it can be directly used in `daline.time` without modification. Additionally, the list of methods should be specified in a cell array, for example, `methodList = {'LS'; 'LS_SVD'; 'LS_COD'}`, where each element represents the name of a built-in linearization method.

Additionally, `daline.time` supports a broad range of method parameters due to its support to measure the time needed by the model training and testing across multiple methods. All parameters relevant to these methods can be integrated into `daline.time`. Users are encouraged to consult Chapter 4 for detailed information on the specific parameters associated with the built-in linearization methods. Furthermore, `daline.time` also leverages the parameters listed in Table 5.2 to tailor the visualization of figures.

Table 5.2: Table of additional parameters specific to `daline.time`

Parameter	Format	Default	Description
<code>PLOT.switch</code>	integer	1	Set to 1 to produce figures; set to 0 otherwise.
<code>PLOT.repeat</code>	integer	3	Specifies the number of times to repeat the process to calculate the average computational time.
<code>PLOT.style</code>	character	'dark'	Select 'dark' to plot the error in a dark environment or 'light' for a light environment. Note: <code>PLOT.style</code> is only valid when <code>PLOT.theme = 'commercial'</code> .

#### Examples

```
>> data = daline.data('case.name', 'case39');
>> method = {'LS'; 'PLS_SIMRX'; 'LS_COD'; 'LS_LIFX'; 'LS_CLS'; 'RR'; 'RR_KPC'; 'RR_WEI'; '
    LS_PIN'; 'LS_PCA'; 'PLS_NIP'; 'PLS_CLS'; 'TAY'; 'DC'; 'PTDF'; 'DLPF'; 'DLPF_C'; 'DC_LS'
    };
>> daline.time(data, method);
% See the output figure below.
% (The data and method list will be used for the following two examples.)
```

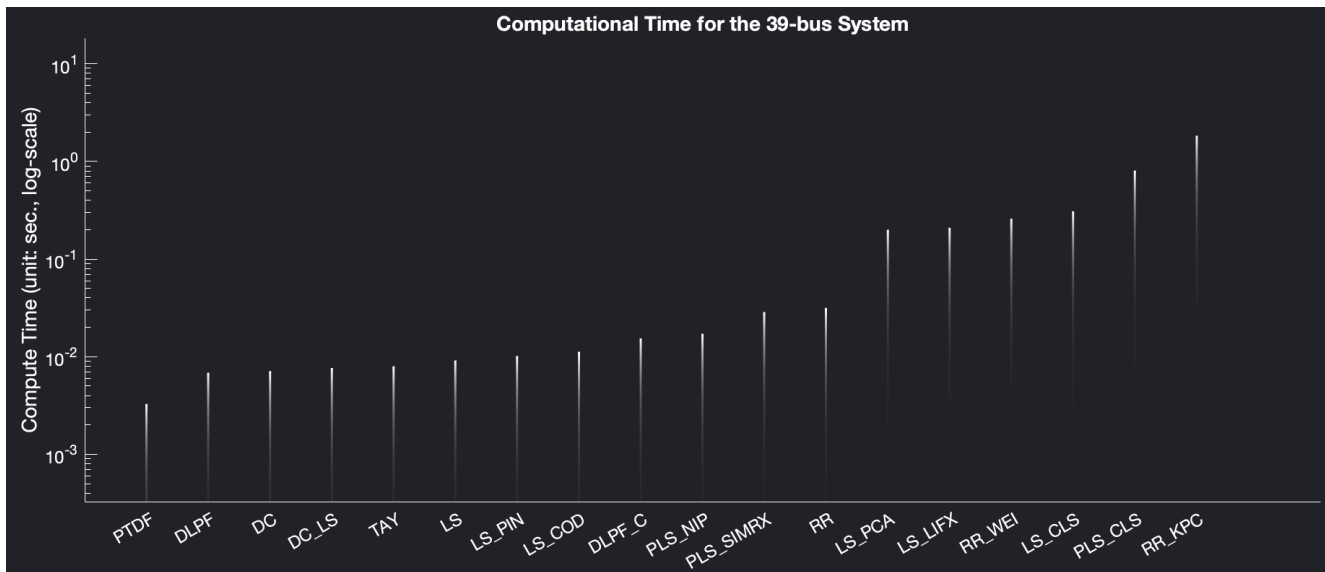


Figure 5.10: The dark theme for the computational time ranking.

```
>> timeList = daline.time(data, method, 'PLOT.repeat', 5, 'PLOT.style', 'light');
% The output 'timeList' is a vector; each element is the computational time of a
% given method in 'method'. The sequence of 'timeList' is identical to the method
% sequence in 'method'.
% See the output figure below.
```

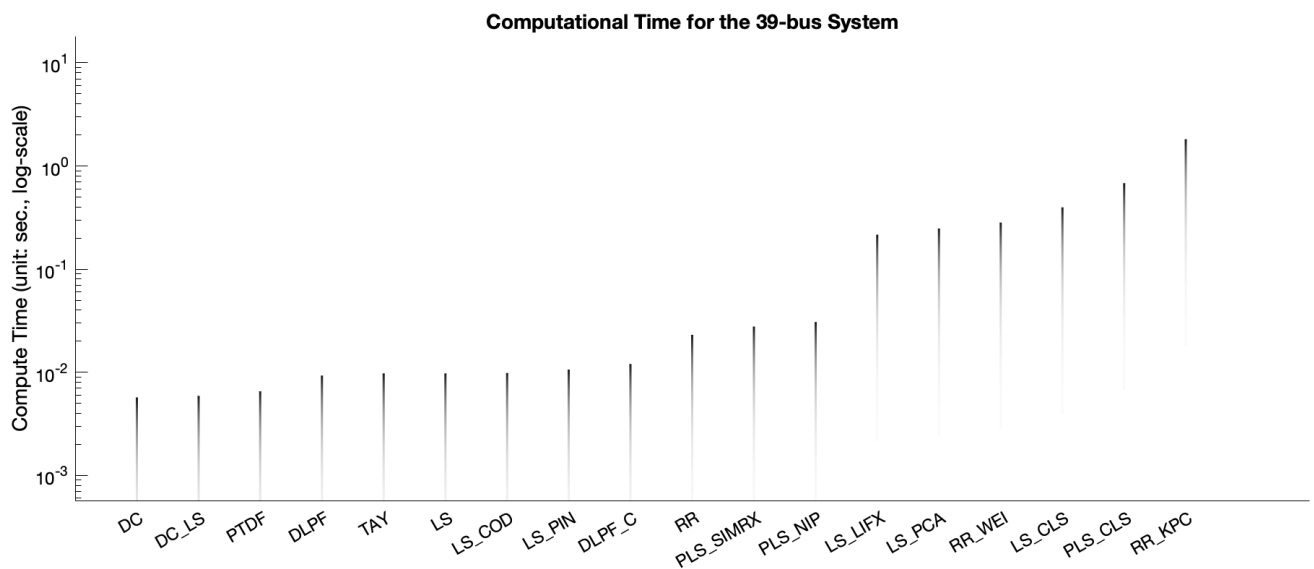


Figure 5.11: The light theme for the computational time ranking.

```
>> opt = daline.setopt('PLOT.repeat', 3, 'PLOT.style', 'dark');
>> timeList = daline.time(data, method, opt);
% Like the other wrappers, daline.rank also accepts the structured 'opt' formulated
% by daline.setopt.
```

*% See the output figure below.*

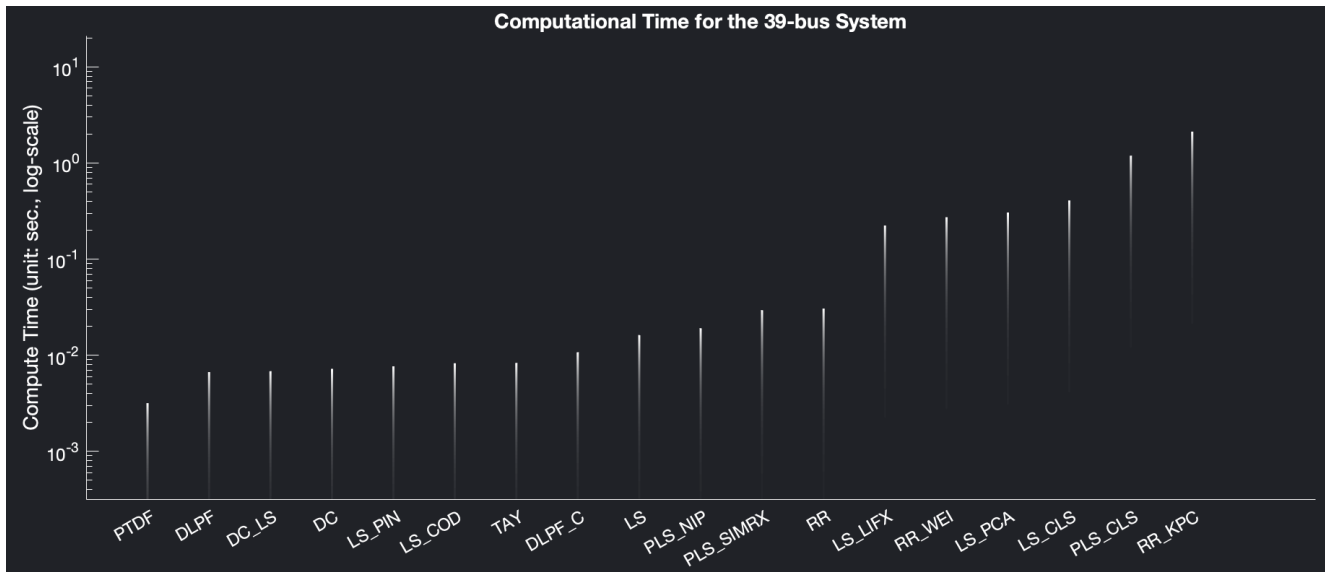


Figure 5.12: The dark theme for the computational time ranking.

## 5.2.2 Computational time evolution curves of multiple methods (daline.time)

### Inputs

When generating computational evolution curves for multiple test systems using various built-in methods, the required inputs for `daline.time` include a `dataList` cell and a list of method names. Unlike the standardized data format defined by DALINE, here `dataList` should be a cell array, where each element is a `data` struct corresponding to a different test system and conforms to the standardized format. This specialized `data` format can be easily produced by DALINE with minimal coding. Additionally, the list of methods should be defined in a cell array format, such as `methodList = {'LS'; 'LS_SVD'; 'LS_COD'}`, where each element denotes the name of a built-in linearization method.

Additionally, as explained above, `daline.time` also supports a broad range of method parameters, as well as the parameters listed in Table 5.2, to tailor not only the visualization of figures but also the model training and testing processes.

### Examples

```
>> caseList = {'case9', 'case14', 'case33bw', 'case39'};
>> dataList = cell(length(caseList), 1);
>> for n = 1:length(caseList)
>> dataList{n} = daline.data('case.name', caseList{n});
>> end
>> method = {'LS'; 'PLS_SIMRX'; 'LS_COD'; 'LS_LIFX'; 'LS_CLS'; 'RR'; 'RR_KPC'; 'RR_WEI'; '
    LS_PIN'; 'LS_PCA'; 'PLS_NIP'; 'PLS_CLS'; 'TAY'; 'DC'; 'PTDF'; 'DLPF'; 'DLPF_C'; 'DC_LS'
    };
```

```
>> timeList = daline.time(dataList, method);
% See the output figure below.
% (The dataList and method list will be used for the following example.)
```

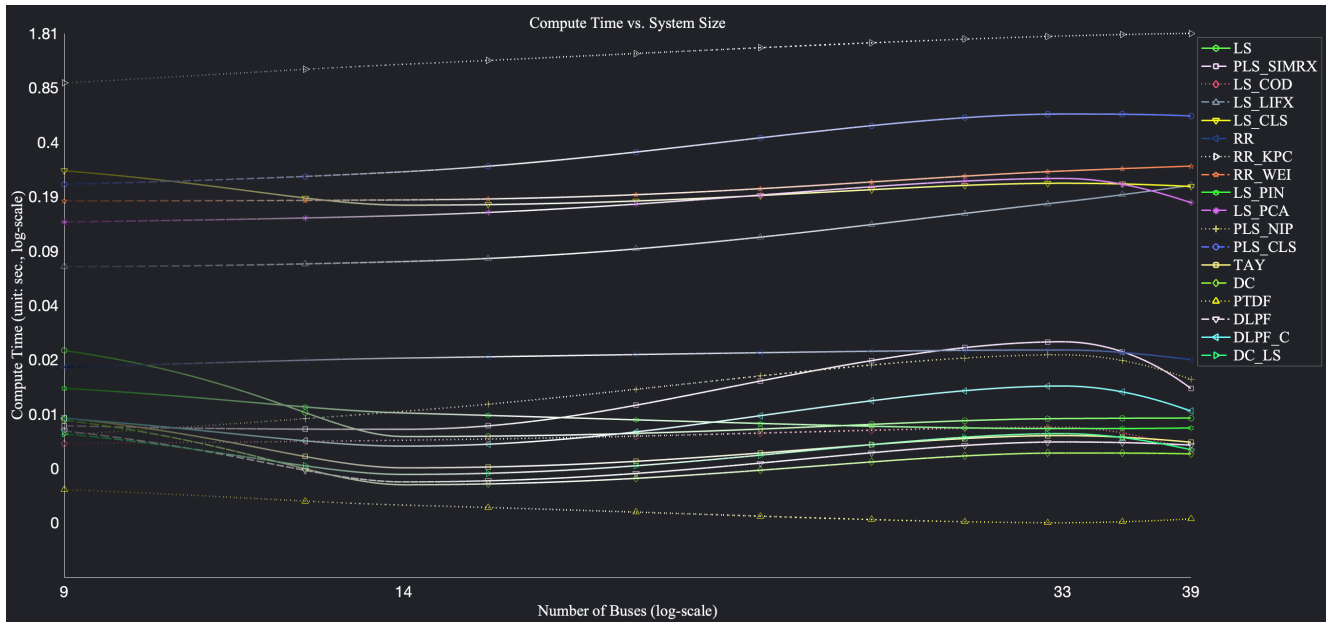


Figure 5.13: The dark theme for the computational evolution curves.

```
>> opt = daline.setopt('variable.predictor', {'P', 'Q'}, 'variable.response', {'PF', 'Vm',
    'RR.lambdaInterval', [0:5e-2:0.1], 'PLOT.repeat', 5, 'PLOT.style', 'light');
>> timeList = daline.time(dataList, method, opt);
% See the output figure below.
```

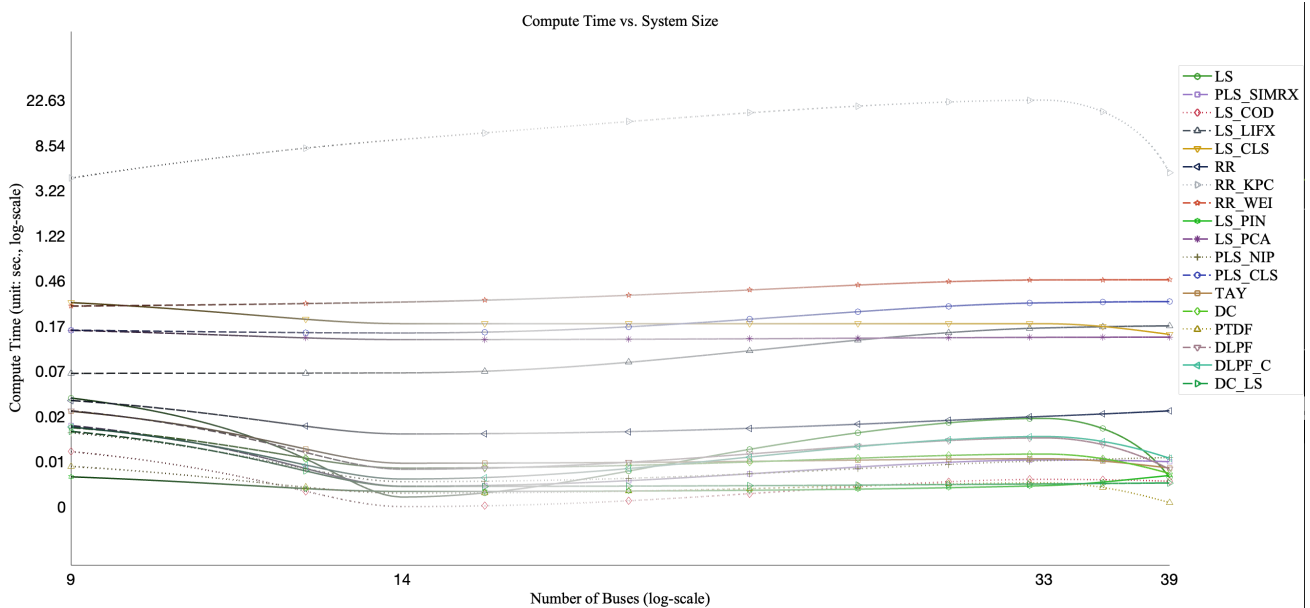


Figure 5.14: The light theme for the computational evolution curves.

## Chapter 6

# All-in-one Command for Daline (`daline.all`)

Minimal coding can greatly enhance the user experience. For certain functionalities — such as data generation, processing, model training, and testing — DALINE provides specialized “all-in-one wrappers” like `daline.data` and `daline.fit`. To further simplify the process, DALINE includes an “all-in-one command” that encompasses most functionalities by bridging `daline.data`, `daline.fit`, and `daline.plot`, enabling users to execute data generation and processing, model training and testing, as well as result visualization with one line of code. This command is `daline.all`.

### 6.1 Inputs

The primary and mandatory input for `daline.all` is a power system case of interest. This can either be a MATPOWER case name, such as `'case118'`, or a power system defined in the standard `mpc` structure. For further details on the `mpc` structure, see Section 3.1 in [36].

Furthermore, `daline.all` supports all parameters specific to `daline.generate`, `daline.noise`, `daline.outlier`, `daline.denoise`, `daline.deoutlier`, `daline.normalize`, `daline.fit`, and `daline.plot`. For information on the built-in parameters of `daline.fit`, refer to Chapter 4; for those of `daline.plot`, see Section ??; and for parameters of the remaining wrappers, consult Chapter 3.

### 6.2 Outputs

`daline.all` provides three optional outputs: `data`, `model`, and `failure`. `data` consists of artificial data generated by DALINE and is structured into the standard data format of DALINE; for details, see Section 3.2.1. `model`, a structure, summarizes the outcomes of model training and testing; for more information, refer to Table 4.3. `failure` captures any potential failure information during the training and testing processes, indicating whether the linearization method encountered any issues when using the generated linear model to compute responses. If no failures occur, `failure` will remain empty.

Additionally, `daline.all` also generates figures that summarize the linearization accuracy of the method from multiple perspectives.

## 6.3 Examples

```
% Example: the minimal input  
>> model = daline.all('case118');
```

```
% Example: the minimal input  
>> mpc = ext2int(loadcase('case118'));  
>> model = daline.all('case118');
```

```
% Example: data generation  
>> model = daline.all('case118', 'num.trainSample', 150, 'num.testSample', 200, 'data.  
parallel', 0, 'data.curvePlot', 1);
```

```
% Examples: adding outliers  
>> model = daline.all('case118', 'num.trainSample', 150, 'num.testSample', 200, '  
outlier.switchTrain', 1);  
>> model = daline.all('case118', 'num.trainSample', 150, 'num.testSample', 200, '  
outlier.switchTrain', 1, 'outlier.switchTest', 1);
```

```
% Examples: adding noise  
>> model = daline.all('case118', 'noise.switchTrain', 1, 'noise.SNR_dB', 30);  
>> model = daline.all('case118', 'noise.switchTest', 1, 'noise.SNR_dB', 30);
```

```
% Examples: filtering outliers  
>> model = daline.all('case118', 'outlier.switchTrain', 1, 'outlier.percentage', 0.01,  
'filOut.switchTrain', 1);  
>> model = daline.all('case118', 'outlier.switchTest', 1, 'outlier.percentage', 0.01,  
'filOut.switchTest', 1, 'filOut.method', 'median');
```

```
% Examples: filtering noise  
>> model = daline.all('case118', 'noise.switchTrain', 1, 'noise.SNR_dB', 30, 'filNoi.  
switchTrain', 1);  
>> model = daline.all('case118', 'noise.switchTrain', 1, 'noise.SNR_dB', 30, 'filNoi.  
switchTrain', 1, 'filNoi.est_dB', 31);
```

```
% Example: normalizing data
```

```
>> [model, data] = daline.all('case118', 'norm.switch', 1);
```

```
% Examples: method settings
```

```
>> [model, data, failure] = daline.all('case118', 'method.name', 'RR');
```

```
>> [model, data, failure] = daline.all('case118', 'method.name', 'RR_KPC', 'RR.  
lambdaInterval', [0.1:0.01:0.2], 'PLOT.switch', 0);
```

```
>> [model, data, failure] = daline.all('case118', 'method.name', 'RR_KPC', 'RR.  
clusNumInterval', 10, 'RR.etaInterval', 1000, 'PLOT.switch', 0);
```

```
>> [model, data, failure] = daline.all('case118', 'method.name', 'RR_VCS');
```

```
>> [model, data, failure] = daline.all('case118', 'method.name', 'RR_VCS', 'RR.  
lambdaInterval', [0.1:0.01:0.2]);
```

```
% Examples: plot settings
```

```
>> [model, data, failure] = daline.all('case118', 'method.name', 'RR_VCS', 'PLOT.  
switch', 0);
```

```
>> [model, data, failure] = daline.all('case118', 'method.name', 'RR_VCS', 'PLOT.  
response', {'Vm', 'PF'}, 'PLOT.theme', 'academic', 'PLOT.pattern', 'indivi');
```

```
% Example: use opt as input
```

```
>> opt = daline.setopt('method.name', 'RR_VCS', 'PLOT.response', {'Vm', 'PF'}, 'PLOT.  
theme', 'academic', 'PLOT.pattern', 'indivi');
```

```
>> [model, data, failure] = daline.all('case118', opt);
```

# Bibliography

- [1] Mengshuo Jia and Gabriela Hug. “Overview of Data-driven Power Flow Linearization”. In: *2023 IEEE Belgrade PowerTech*. 2023, pp. 01–06. DOI: [10.1109/PowerTech55446.2023.10202779](https://doi.org/10.1109/PowerTech55446.2023.10202779) (cit. on p. 1).
- [2] Mengshuo Jia, Gabriela Hug-Glanzmann, Ning Zhang, Zhaojian Wang, Yi Wang, and Chongqing Kang. “Data-driven Power Flow Linearization: Theory”. In: *ETH Zurich Research Collection* (2024). URL: <https://www.research-collection.ethz.ch/handle/20.500.11850/679040> (cit. on pp. 1, 33, 37, 39, 55, 63, 65).
- [3] Mengshuo Jia, Gabriela Hug, Ning Zhang, Zhaojian Wang, Yi Wang, and Chongqing Kang. “Data-driven Power Flow Linearization: Simulation”. In: *arXiv preprint arXiv:2406.06833* (2024). URL: <https://arxiv.org/abs/2406.06833> (cit. on pp. 1, 2, 32, 37, 39, 55, 63, 65, 127).
- [4] Mengshuo Jia, Wen Yi Chan, and Gabriela Hug. “Daline: A Data-driven Power Flow Linearization Toolbox for Power Systems Research and Education”. In: Under Review (2024) (cit. on pp. 2, 4).
- [5] *The BSD 3-Clause License*. <http://opensource.org/licenses/BSD-3-Clause>. Open Source Initiative (cit. on p. 3).
- [6] Ray Daniel Zimmerman, Carlos Edmundo Murillo-Sánchez, and Robert John Thomas. “MATPOWER: Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education”. In: *IEEE Transactions on Power Systems* 26.1 (2011), pp. 12–19. DOI: [10.1109/TPWRS.2010.2051168](https://doi.org/10.1109/TPWRS.2010.2051168) (cit. on p. 3).
- [7] Inc. CVX Research. *CVX: Matlab Software for Disciplined Convex Programming, version 2.0*. <http://cvxr.com/cvx>. Aug. 2012 (cit. on p. 3).
- [8] J. Löfberg. “YALMIP : A Toolbox for Modeling and Optimization in MATLAB”. In: *In Proceedings of the CACSD Conference*. Taipei, Taiwan, 2004 (cit. on p. 3).
- [9] Hanchen Xu, Alejandro D Domínguez-García, Venugopal V Veeravalli, and Peter W Sauer. “Data-driven voltage regulation in radial power distribution systems”. In: *IEEE Transactions on Power Systems* 35.3 (2019), pp. 2133–2143 (cit. on pp. 10, 61, 63).



- [10] Zhentong Shao, Qiaozhu Zhai, Jiang Wu, and Xiaohong Guan. “Data Based Linear Power Flow Model: Investigation of a Least-Squares Based Approximation”. In: *IEEE Transactions on Power Systems* 36.5 (2021), pp. 4246–4258 (cit. on pp. 10, 41, 42).
- [11] Shao Zhentong, Zhai Qiaozhu, Wu Jiang, and Guan Xiaohong. “Data Based Linearization: Least-Squares Based Approximation”. In: *arXiv preprint arXiv:2007.02494* (2020) (cit. on p. 10).
- [12] Yitong Liu, Zhengshuo Li, and Yu Zhou. “Data-Driven-Aided Linear Three-Phase Power Flow Model for Distribution Power Systems”. In: *IEEE Transactions on Power Systems* (2021) (cit. on p. 10).
- [13] Yitong Liu, Zhengshuo Li, and Yu Zhou. “A Physics-based and Data-driven Linear Three-Phase Power Flow Model for Distribution Power Systems”. In: *arXiv preprint arXiv:2103.10147* (2021) (cit. on p. 10).
- [14] Yuxiao Liu, Yi Wang, Ning Zhang, Dan Lu, and Chongqing Kang. “A data-driven approach to linearize power flow equations considering measurement noise”. In: *IEEE Transactions on Smart Grid* 11.3 (2019), pp. 2576–2587 (cit. on pp. 10, 101).
- [15] Siobhan Powell, Alyona Ivanova, and David Chassin. “Fast solutions in power system simulation through coupling with data-driven power flow models for voltage estimation”. In: *arXiv preprint arXiv:2001.01714* (2020) (cit. on p. 10).
- [16] Li Guo, Yuxuan Zhang, Xialin Li, Zhongguan Wang, Yixin Liu, Linqun Bai, and Chengshan Wang. “Data-driven Power Flow Calculation Method: A Lifting Dimension Linear Regression Approach”. In: *IEEE Transactions on Power Systems* (2021) (cit. on pp. 10, 58–60).
- [17] Milan Korda and Igor Mezić. “Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control”. In: *Automatica* 93 (2018), pp. 149–160 (cit. on pp. 10, 56, 58, 60).
- [18] Yitong Liu, Zhengshuo Li, and Shumin Sun. “A Data-Driven Method for Online Constructing Linear Power Flow Model”. In: *IEEE Transactions on Industry Applications* (2023) (cit. on pp. 10, 26).
- [19] Yi Tan, Yuanyang Chen, Yong Li, and Yijia Cao. “Linearizing power flow model: A hybrid physical model-driven and data-driven approach”. In: *IEEE Transactions on Power Systems* 35.3 (2020), pp. 2475–2478 (cit. on p. 10).
- [20] Yuxiao Liu, Ning Zhang, Yi Wang, Jingwei Yang, and Chongqing Kang. “Data-driven power flow linearization: A regression approach”. In: *IEEE Transactions on Smart Grid* 10.3 (2018), pp. 2569–2580 (cit. on pp. 10, 67, 72).
- [21] Severin Nowak, Yu Christine Chen, and Liwei Wang. “Measurement-based optimal DER dispatch with a recursively estimated sensitivity model”. In: *IEEE Transactions on Power Systems* 35.6 (2020), pp. 4792–4802 (cit. on pp. 10, 75).
- [22] Yanbo Chen, Chao Wu, and Junjian Qi. “Data-Driven Power Flow Method Based on Exact Linear Regression Equations”. In: *Journal of Modern Power Systems and Clean Energy* (2021) (cit. on p. 10).

- [23] Jiaqi Chen, Wenchuan Wu, and Line A Roald. “Data-driven Piecewise Linearization for Distribution Three-phase Stochastic Power Flow”. In: *IEEE Transactions on Smart Grid* (2021) (cit. on p. 10).
- [24] Junbo Zhang, Zejing Wang, Xiangtian Zheng, Lin Guan, and CY Chung. “Locally weighted ridge regression for power system online sensitivity identification considering data collinearity”. In: *IEEE Transactions on Power Systems* 33.2 (2017), pp. 1624–1634 (cit. on p. 10).
- [25] Jiaqi Chen, Wenyun Li, Wenchuan Wu, Tao Zhu, Zhenyi Wang, and Chuan Zhao. “Robust Data-driven Linearization for Distribution Three-phase Power Flow”. In: *2020 IEEE 4th Conference on Energy Internet and Energy System Integration (EI2)*. IEEE. 2020, pp. 1527–1532 (cit. on p. 10).
- [26] Zhentong Shao, Qiaozhu Zhai, Zhihan Han, and Xiaohong Guan. “A linear AC unit commitment formulation: An application of data-driven linear power flow model”. In: *International Journal of Electrical Power & Energy Systems* 145 (2023), p. 108673 (cit. on pp. 10, 114).
- [27] Jiafan Yu, Yang Weng, and Ram Rajagopal. “Robust mapping rule estimation for power flow analysis in distribution grids”. In: *2017 North American Power Symposium (NAPS)*. IEEE. 2017, pp. 1–6 (cit. on p. 10).
- [28] Jiafan Yu, Yang Weng, and Ram Rajagopal. “Mapping rule estimation for power flow analysis in distribution grids”. In: *arXiv preprint arXiv:1702.07948* (2017) (cit. on p. 10).
- [29] Penghua Li, Wenchuan Wu, Xiaoming Wan, and Bin Xu. “A Data-Driven Linear Optimal Power Flow Model for Distribution Networks”. In: *IEEE Transactions on Power Systems* (2022) (cit. on pp. 10, 94).
- [30] Yuxiao Liu, Bolun Xu, Audun Botterud, Ning Zhang, and Chongqing Kang. “Bounding regression errors in data-driven power grid steady-state models”. In: *IEEE Transactions on Power Systems* 36.2 (2020), pp. 1023–1033 (cit. on p. 10).
- [31] Yitong Liu, Zhengshuo Li, and Junbo Zhao. “Robust Data-Driven Linear Power Flow Model With Probability Constrained Worst-Case Errors”. In: *IEEE Transactions on Power Systems* 37.5 (2022), pp. 4113–4116. DOI: [10.1109/TPWRS.2022.3189543](https://doi.org/10.1109/TPWRS.2022.3189543) (cit. on pp. 10, 110, 112, 114).
- [32] Xingpeng Li. “Fast Heuristic AC Power Flow Analysis with Data-Driven Enhanced Linearized Model”. In: *Energies* 13.13 (2020), p. 3308 (cit. on p. 10).
- [33] Xingpeng Li and Kory Hedman. “Data driven linearized ac power flow model with regression analysis”. In: *arXiv preprint arXiv:1811.09727* (2018) (cit. on p. 10).
- [34] Mengshuo Jia, Qianni Cao, Chen Shen, and Gabriela Hug. “Frequency-Control-Aware Probabilistic Load Flow: An Analytical Method”. In: *IEEE Transactions on Power Systems* (2022), pp. 1–16. DOI: [10.1109/TPWRS.2022.3223884](https://doi.org/10.1109/TPWRS.2022.3223884) (cit. on p. 10).
- [35] Jingwei Yang, Ning Zhang, Chongqing Kang, and Qing Xia. “A state-independent linear power flow model with accurate estimation of voltage magnitude”. In: *IEEE Transactions on Power Systems* 32.5 (2016), pp. 3607–3617 (cit. on p. 10).
- [36] Ray D Zimmerman and Carlos E Murillo-Sánchez. “Matpower 6.0 user’s manual”. In: *Power Systems Engineering Research Center* 9 (2016) (cit. on pp. 17, 22, 143).

- [37] Michael Brown, Milan Biswal, Sukumar Brahma, Satish J Ranade, and Huiping Cao. “Characterizing and quantifying noise in PMU data”. In: *2016 IEEE Power and Energy Society General Meeting (PESGM)*. IEEE. 2016, pp. 1–5 (cit. on p. 23).
- [38] Kebina Manandhar, Xiaojun Cao, Fei Hu, and Yao Liu. “Detection of Faults and Attacks Including False Data Injection Attack in Smart Grid Using Kalman Filter”. In: *IEEE Transactions on Control of Network Systems* 1.4 (2014), pp. 370–379. DOI: [10.1109/TCNS.2014.2357531](https://doi.org/10.1109/TCNS.2014.2357531) (cit. on p. 26).
- [39] Li Guo, Yuxuan Zhang, Xialin Li, Zhongguan Wang, Yixin Liu, Linqun Bai, and Chengshan Wang. “Data-driven Power Flow Calculation Method: A Lifting Dimension Linear Regression Approach”. In: *IEEE Transactions on Power Systems* (2021) (cit. on pp. 39, 40, 58).
- [40] Yitong Liu, Zhengshuo Li, and Yu Zhou. “Data-Driven-Aided Linear Three-Phase Power Flow Model for Distribution Power Systems”. In: *IEEE Transactions on Power Systems* (2021) (cit. on pp. 46, 49).
- [41] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004 (cit. on p. 49).
- [42] The MathWorks Inc. *Feasible generalized least squares*. 2023. URL: <https://mathworks.com/help/econ/fglsl.html#buicqm5-17> (cit. on p. 52).
- [43] Carl Mugnier, Konstantina Christakou, Joel Jatton, Michael De Vivo, Mauro Carpita, and Mario Paolone. “Model-less/measurement-based computation of voltage sensitivities in unbalanced electrical distribution networks”. In: *2016 Power Systems Computation Conference (PSCC)*. IEEE. 2016, pp. 1–7 (cit. on p. 52).
- [44] Yuxiao Liu, Yi Wang, Ning Zhang, Dan Lu, and Chongqing Kang. “A data-driven approach to linearize power flow equations considering measurement noise”. In: *IEEE Transactions on Smart Grid* 11.3 (2019), pp. 2576–2587 (cit. on p. 54).
- [45] Yitong Liu, Zhengshuo Li, and Shumin Sun. “A Data-Driven Method for Online Constructing Linear Power Flow Model”. In: *IEEE Transactions on Industry Applications* (2023) (cit. on pp. 64, 66).
- [46] Manoj Badoni, Alka Singh, and Bhim Singh. “Variable Forgetting Factor Recursive Least Square Control Algorithm for DSTATCOM”. In: *IEEE Transactions on Power Delivery* 30.5 (2015), pp. 2353–2361. DOI: [10.1109/TPWRD.2015.2422139](https://doi.org/10.1109/TPWRD.2015.2422139) (cit. on pp. 64, 66).
- [47] S Joe Qin. “Partial least squares regression for recursive system identification”. In: *Proceedings of 32nd IEEE Conference on Decision and Control*. IEEE. 1993, pp. 2617–2622 (cit. on pp. 67, 70).
- [48] Aylin Alin. “Comparison of PLS algorithms when number of objects is much larger than number of variables”. In: *Statistical papers* 50.4 (2009), pp. 711–720 (cit. on pp. 67, 68).
- [49] Yi Tan, Yuanyang Chen, Yong Li, and Yijia Cao. “Linearizing power flow model: A hybrid physical model-driven and data-driven approach”. In: *IEEE Transactions on Power Systems* 35.3 (2020), pp. 2475–2478 (cit. on p. 68).

- [50] Sijmen De Jong. “SIMPLS: an alternative approach to partial least squares regression”. In: *Chemometrics and intelligent laboratory systems* 18.3 (1993), pp. 251–263 (cit. on p. 68).
- [51] Herman Wold. “Path models with latent variables: The NIPALS approach”. In: *Quantitative sociology*. Elsevier, 1975, pp. 307–357 (cit. on p. 70).
- [52] Yuxiao Liu, Ning Zhang, Yi Wang, Jingwei Yang, and Chongqing Kang. “Data-driven power flow linearization: A regression approach”. In: *IEEE Transactions on Smart Grid* 10.3 (2018), pp. 2569–2580 (cit. on pp. 71, 73).
- [53] S Joe Qin. “Recursive PLS algorithms for adaptive data modeling”. In: *Computers & Chemical Engineering* 22.4-5 (1998), pp. 503–514 (cit. on p. 75).
- [54] Severin Nowak, Yu Christine Chen, and Liwei Wang. “Measurement-based optimal DER dispatch with a recursively estimated sensitivity model”. In: *IEEE Transactions on Power Systems* 35.6 (2020), pp. 4792–4802 (cit. on p. 77).
- [55] Yanbo Chen, Chao Wu, and Junjian Qi. “Data-Driven Power Flow Method Based on Exact Linear Regression Equations”. In: *Journal of Modern Power Systems and Clean Energy* (2021) (cit. on pp. 82, 84).
- [56] Jiaqi Chen, Wenchuan Wu, and Line A Roald. “Data-driven Piecewise Linearization for Distribution Three-phase Stochastic Power Flow”. In: *IEEE Transactions on Smart Grid* (2021) (cit. on p. 86).
- [57] Junbo Zhang, Zejing Wang, Xiangtian Zheng, Lin Guan, and CY Chung. “Locally weighted ridge regression for power system online sensitivity identification considering data collinearity”. In: *IEEE Transactions on Power Systems* 33.2 (2017), pp. 1624–1634 (cit. on p. 87).
- [58] Alex J Smola and Bernhard Schölkopf. “A tutorial on support vector regression”. In: *Statistics and computing* 14.3 (2004), pp. 199–222 (cit. on pp. 89, 91).
- [59] Jiaqi Chen, Wenyun Li, Wenchuan Wu, Tao Zhu, Zhenyi Wang, and Chuan Zhao. “Robust Data-driven Linearization for Distribution Three-phase Power Flow”. In: *2020 IEEE 4th Conference on Energy Internet and Energy System Integration (EI2)*. IEEE. 2020, pp. 1527–1532 (cit. on p. 90).
- [60] Jiafan Yu, Yang Weng, and Ram Rajagopal. “Robust mapping rule estimation for power flow analysis in distribution grids”. In: *2017 North American Power Symposium (NAPS)*. IEEE. 2017, pp. 1–6 (cit. on p. 92).
- [61] Penghua Li, Wenchuan Wu, Xiaoming Wang, and Bin Xu. “A Data-Driven Linear Optimal Power Flow Model for Distribution Networks”. In: *IEEE Transactions on Power Systems* (2022) (cit. on p. 94).
- [62] Zhentong Shao, Qiaozhu Zhai, Zhihan Han, and Xiaohong Guan. “A linear AC unit commitment formulation: An application of data-driven linear power flow model”. In: *International Journal of Electrical Power & Energy Systems* 145 (2023), p. 108673 (cit. on pp. 96, 114).
- [63] Yuxiao Liu, Bolun Xu, Audun Botterud, Ning Zhang, and Chongqing Kang. “Bounding regression errors in data-driven power grid steady-state models”. In: *IEEE Transactions on Power Systems* 36.2 (2020), pp. 1023–1033 (cit. on pp. 99, 100).

- [64] Yuxiao Liu, Yi Wang, Ning Zhang, Dan Lu, and Chongqing Kang. “A data-driven approach to linearize power flow equations considering measurement noise”. In: *IEEE Transactions on Smart Grid* 11.3 (2019), pp. 2576–2587 (cit. on pp. 101, 102).
- [65] Yuxiao Liu, Bolun Xu, Audun Botterud, Ning Zhang, and Chongqing Kang. “Bounding regression errors in data-driven power grid steady-state models”. In: *IEEE Transactions on Power Systems* 36.2 (2020), pp. 1023–1033 (cit. on pp. 104, 105).
- [66] Yiling Zhang, Siqian Shen, and Johanna L Mathieu. “Distributionally robust chance-constrained optimal power flow with uncertain renewables and uncertain reserves provided by loads”. In: *IEEE Transactions on Power Systems* 32.2 (2016), pp. 1378–1388 (cit. on pp. 108, 110, 112).
- [67] Yiling Zhang, Siqian Shen, and Johanna L Mathieu. “Distributionally robust chance-constrained optimal power flow with uncertain renewables and uncertain reserves provided by loads”. In: *IEEE Transactions on Power Systems* 32.2 (2016), pp. 1378–1388 (cit. on p. 109).
- [68] Yitong Liu, Zhengshuo Li, and Junbo Zhao. “Robust Data-Driven Linear Power Flow Model with Probability Constrained Worst-Case Errors”. In: *arXiv preprint arXiv:2112.10320* (2021) (cit. on p. 112).
- [69] Wei Wei. “Tutorials on Advanced Optimization Methods”. In: *arXiv preprint arXiv:2007.13545* (2020) (cit. on p. 113).
- [70] Wei Wei. “Tutorials on Advanced Optimization Methods”. In: *arXiv preprint arXiv:2007.13545* (2020) (cit. on p. 114).
- [71] Ray D Zimmerman and Carlos E Murillo-Sánchez. “Matpower 6.0 user’s manual”. In: *Power Systems Engineering Research Center* 9 (2016) (cit. on pp. 115, 118).
- [72] Xingpeng Li and Kory Hedman. “Data driven linearized ac power flow model with regression analysis”. In: *arXiv preprint arXiv:1811.09727* (2018) (cit. on pp. 116, 120).
- [73] Jingwei Yang, Ning Zhang, Chongqing Kang, and Qing Xia. “A state-independent linear power flow model with accurate estimation of voltage magnitude”. In: *IEEE Transactions on Power Systems* 32.5 (2016), pp. 3607–3617 (cit. on p. 117).
- [74] Mengshuo Jia, Qianni Cao, Chen Shen, and Gabriela Hug. “Frequency-Control-Aware Probabilistic Load Flow: An Analytical Method”. In: *IEEE Transactions on Power Systems* 38.6 (2023), pp. 5170–5187 (cit. on p. 118).