EP-BOLFI

Release 3.0

Yannick Kuhn

Jan 24, 2025

CONTENTS

CHAPTER

ONE

README

EP-BOLFI

EP-BOLFI (Expectation Propagation with Bayesian Optimization for Likelihood-Free Inference) extends the simulator optimizer BOLFI with the data featurization of Expectation Propagation. EP-BOLFI inherits the advantages of both: high stability to measurement noise and considerable reduction of computational effort. The performance is one to two orders of magnitude better than Markov-Chain Monte Carlo, counted in the number of simulator evaluations required.

Documentation

ep_bolfi contains the EP-BOLFI algorithm, a few example models, and utility functions that cover fitting functions, datafile imports, dataset processing, and visualization. For further details, please refer to [the HTML documentation on GitLab Pages](https://cec.pages.gitlab.dlr.de/bte/ep-bolfi/).

Examples

The examples currently comprise of the code used in the [EP-BOLFI publication](https://doi.org/10.48550/arXiv.2208.03289). Apart from the SPMe benchmark, they analyze a GITT dataset provided by BASF. You can find the dataset at the DOI [10.5281/zenodo.7478267](https://doi.org/10.5281/zenodo.7478267). If you wish to re-run the parameterization yourself, copy the contents of GITT_data_and_parameterization_info.zip into examples first.

- To perform the SPMe benchmarks from Aitio et al., please use spme_benchmark_multimodal.py and spme_benchmark_unimodal.py. To calculate the tabulated results presented in the article, use evaluate_spme_benchmark_confidence.py.
- To preprocess the OCV curves, please use ocv_from_cccv_and_gitt.py.
- To view the GITT data, please use measurement_plot.py.
- To estimate parameters from the GITT data from BASF, use run_estimation.py, and after that collect_gitt_estimation_results.py. This may take more than a week to run.
- To get the tabulated results for the GITT parameterization, please use evaluate_gitt_estimation_confidence.py.
- To perform the verification of the GITT data parameterization, please use verify_estimation.py. This may take more than a week to run.
- To analyze the correlation matrix for the 7-parameter estimation, please use correlation_visualization.py.
- To plot the results from the GITT estimation procedure, please use gitt_visualization.py.
- To analyze the precision and reliability of the GITT estimation procedure, please use calculate_experimental_and_simulated_features.py. To plot this analysis, please use analytic_vs_epbolfi_results.py and sensitivity_visualization.py.
- To plot the joint resistance of the two exchange-current densities, please use joint_resistance.py.

Using EP-BOLFI to process your measurements with your model

Please have a look at [the setup example](examples/parameters/estimation/). [gitt_basf.py](examples/parameters/estimation/gitt_basf.py) contains the preprocessing of the GITT dataset and a GITT simulator. [gitt.py](examples/parameters/estimation/gitt.py) contains one possible definition of features in a GITT measurement. For performing the optimization, please have a look at run_estimation.py. If you wish to re-use an optimization result as starting value, use .Q, .r, .Q_features and .r_features and pass them to the initialization of the EP-BOLFI object.

Installation

EP-BOLFI requires [Python 3.8 or greater](https://www.python.org/downloads/). Just download the newest Release .whl file and requirements.txt. Install the dependencies with requirements.txt via [pip](https://pypi.org/project/pip/), then install the .whl file. In case you want to build the package from source, please refer to [CONTRIBUTING.md](CONTRIBUTING.md#building-from-source).

Using pip

Create a virtual environment and activate it. On Linux and Mac: `bash python3.9 -m venv ep_bolfi source ep_bolfi/bin/activate ` On Windows: `powershell py -3.9 -m venv ep_bolfi . .\ep_bolfi\Scripts\activate `

Then install the dependencies and package: `bash pip install -r requirements.txt pip install ep_bolfi-${VERSION}-py3-none-any.whl `

Using [miniforge](https://github.com/conda-forge/miniforge)

On Windows, first initialize conda to work with PowerShell, assuming you work with PowerShell: `powershell conda init powershell `

Create a new virtual environment: `bash conda create -n ep_bolfi python=3.9 conda activate ep_bolfi ` and install the dependencies there. The fastest way is to use pip to install the packages: `bash pip install -r requirements.txt ` If you want to use the conda package manager as much as possible, use these lines instead: `bash conda env update --file ep_bolfi.yml pip install -r pip_only_requirements.txt `

Then install the dependencies and package: `bash conda install ep_bolfi-${VERSION}-py39_0.tar.bz2 `

Using the Kadi4Mat tools

The files in (ep_bolfi/kadi_tools) are command-line tools. These command-line tools are designed to interface with the database software [Kadi4Mat](https://kadi.iam-cms.kit.edu/), of which we have an internal instance running at (https://kadi-dlr.hiu-batteries.de/).

In order to use these command-line tools in the workflow toolchain of Kadi4Mat, they are implemented with the library [xmlhelpy](https://gitlab.com/iam-cms/workflows/xmlhelpy) which extends the [Click](https://github.com/pallets/click) library for command-line tools with the option to generate machine-readable representations of command-line tools. You can find these representations in *ep_bolfi/kadi_tools/xml_representations*, but only when installing a Release. If they are missing in your installation, please refer to the manual instructions in CONTRIBUTING.md.

[KadiStudio](https://bwsyncandshare.kit.edu/s/cJSZrE6fDTR6cLQ) can import the .py files, or on a Kadi4Mat instance with the online workflow editor enabled, the .xml files can be imported by uploading them to any Record on the respective Kadi4Mat instance. The command-line tools are then available as building blocks for workflows.

For executing workflows that contain one of these tools, the command *python* has to launch a Python environment with this library installed. Either bake the activation of said environment into the convenience scripts in the following section, or make your command line automatically activate it by following these steps.

On Linux: `bash nano ~/.profile ` Then add the following line to the bottom of the file: ` source ~/ep_bolfi/bin/activate `

On Windows: `powershell notepad $((Split-Path $profile -Parent) + "\profile.ps1") ` Then add the following line to the bottom of the text file: ` . .\ep_bolfi\Scripts\activate `

In the case where spurious lines like “warning in …: failed to import cython module: falling back to numpy” show up and break workflow scripts, these are due to an unfortunate design decision in GPy. You need to install GPy from source like so to improve performance as well:

`bash git clone https://github.com/SheffieldML/GPy.git cd GPy pip install . `

For the smoothest installation experience, we recommend using the HPC JUSTUS 2.

Convenience scripts to execute Kadi4Mat workflows

Since [KadiStudio](https://bwsyncandshare.kit.edu/s/cJSZrE6fDTR6cLQ) is semi-usable at this point in time to work with Kadi4Mat workflows, there are two helper scripts in this repository for executing workflows automatically, including input interactions. First, install the libraries [process engine](https://gitlab.com/iam-cms/workflows/process-engine) and [process manager](https://gitlab.com/iam-cms/workflows/process-manager). We recommend Linux or using Linux indirectly via [WSL](https://learn.microsoft.com/en-us/windows/wsl/install) on Windows. On Linux, (ep_bolfi/utility/execute_workflow_locally.py) can be made into an executable with [PyInstaller](https://pyinstaller.org/en/stable/) that can be registered with your OS or browser to execute workflow files:

`bash pip install pyinstaller pyinstaller ep_bolfi/utility/execute_workflow_locally.py `

You then find the executable inside (dist/execute_workflow_locally/). Do not use (build), and move the whole folder (dist/execute_workflow_locally/) to where you want to store executable, as the executable depends on the folder contents.

On Windows, use (ep_bolfi/utility/send_workflow_to_wsl.py). It is a wrapper for the case where the process engine and process manager are installed within a WSL on Windows. First, make sure that you installed EP-BOLFI as described above first within WSL. The script assumes you did so within a virtual environment ~/venv in WSL, like so:

`bash pip install virtualenv cd ~ mkdir venv virtualenv venv source venv/bin/activate pip/conda install ... `

Then, instead of (ep_bolfi/utility/execute_workflow_locally.py), you make an executable out of (ep_bolfi/utility/send_workflow_to_wsl.py) in Windows:

`cmd pip install pyinstaller pyinstaller ep_bolfi/utility/send_workflow_to_wsl.py `

Then move the folder (dist/send_workflow_to_wsl/) to where you want to store the executable and then register the contained (send_workflow_to_wsl.exe) with your OS or browser to open workflow files. Remember to install and activate [Xming](http://www.straightrunning.com/XmingNotes/changes.php) if you wish to have graphical outputs.

Contributing to EP-BOLFI

Please refer to the [contributing guidelines](CONTRIBUTING.md) and adhere to the [Code of Conduct](CODE_OF_CONDUCT.md).

Licence

This is a purely internal repository, hence there is no licence. Please note that the lack of a licence bars you from using this code in any way, shape or form outside CEC. Please use the code published on https://github.com/YannickNoelStephanKuhn/EP-BOLFI for those purposes instead.

CHAPTER

TWO

API

ep_bolfi

2.1 ep_bolfi

Modules

kadi_tools	Each file in here is a CLI-executable tool, in the format:
models	
optimization	
utility	

2.1.1 ep_bolfi.kadi_tools

Each file in here is a CLI-executable tool, in the format:

- python -m ep_bolfi.kadi_tools.<**filename_without_**.py>

Due to the way the CLI-executableness is implemented, the methods are not directly callable the way they appear in the source code, so please refer to their –help outputs, which are stored alongside documentation.

If you want to use the methods from inside a Python script, you need to imitate the CLI execution, for example:

from subprocess import run from ep_bolfi.utility.dataset_formatting import Cycling_Information read_call = run([

> “python”, “-m”, “ep_bolfi.kadi_tools.read_measurement_from_parquet”, “-r”, “<record_id>”, “-n”, “<file-name>”,

], capture_output=True) data = Cycling_Information.from_json(read_call.stdout)

Modules

collect_and_fit_spline_to_parameterization_points	Reads in multiple parameterization result files representing individual data points of a function dependent on some independent variable, and fits a spline to data points given their independent variable.
combine_measurement_segments	Reads in a JSON representation of a `Measurement` object, combines segments of it as per the given rules, and outputs the resulting `Measurement` object in a JSON representation again.
convert_csv_to_parquet	Stores CSV file contents in an Apache Parquet file.
eis_parameterization	Reads in an input file prepared by `ep_bolfi.kadi_tools.eis_preprocessing` and runs EP-BOLFI on it.
eis_preprocessing	Reads in an `Impedance_Measurement` json representation and preprocesses it as EIS data, assuming each segment corresponds to an equilibrium.
eis_visualization	Reads in an input file prepared by `ep_bolfi.kadi_tools.eis_preprocessing` and runs and compares the simulations for the initial parameter guesses against the data.
extract_measurement_protocol	Reads in a `Cycling_Information` json representation and outputs a text-based representation of the different segments that corresponds to `pybamm.Experiment` inputs.
extract_ocv_curve	Reads in a `Cycling_Information` json representation and extracts OCV information from it, assuming it represents a GITT measurement.
extract_overpotential	Reads in a `Cycling_Information` json representation and applies a PyBaMM model parameter set to subtract the OCP curve(s) from it, leaving us with only the overpotential.
fit_and_plot_ocv	Reads in a json file containing OCV information and fits the model from Birkl2015 (A Parametric OCV Model for Li-Ion Batteries, DOI 10.1149/2.0331512jes) to it.
fit_exponential_decay	Reads in a `Cycling_Information` json representation and fits an exponential fit function to its first segment.
gitt_parameterization	Reads in a `Cycling_Information` json representation and uses it in conjunction with input files generated by `ep_bolfi.kadi_tools.gitt_preprocessing` to fit any PyBaMM model to it via EP-BOLFI.
gitt_preprocessing	Reads in a `Cycling_Information` json representation and preprocesses it as GITT data, assuming each segment corresponds to a pulse or pause.
gitt_visualization	Reads in an input file prepared by `ep_bolfi.kadi_tools.gitt_preprocessing` and runs and compares the simulations for the initial parameter guesses against the data.
plot_measurement	Reads in a `Cycling_Information` json representation and plots it.
read_csv_datasets	Preprocesses commonly encountered CSV measurement files to the datatypes defined in `ep_bolfi.utility.dataset_formatting`.
read_hdf5_datasets	Provides preprocessing from HDF5 contents to the datatypes defined in `ep_bolfi.utility.dataset_formatting`.
read_measurement_from_parquet	Reads a `Measurement` object that got stored in an Apache Parquet file with `ep_bolfi.kadi_tools.store_measurement_in_parquet`.

ep_bolfi.kadi_tools.collect_and_fit_spline_to_parameterization_points

Reads in multiple parameterization result files representing individual data points of a function dependent on some independent variable, and fits a spline to data points given their independent variable.

Functions

get_valid_filename(s)	Takes a string and transforms it into an ASCII-only filename.

ep_bolfi.kadi_tools.collect_and_fit_spline_to_parameterization_points.get_valid_filename

ep_bolfi.kadi_tools.collect_and_fit_spline_to_parameterization_points.**get_valid_filename**(*s*)

Takes a string and transforms it into an ASCII-only filename.

Parameters
s – Any string.

Returns
s, with diacritics stripped and non-letters removed except -_. Multiple adjacent instances of - or _ are shortened to one.

ep_bolfi.kadi_tools.combine_measurement_segments

Reads in a JSON representation of a `Measurement` object, combines segments of it as per the given rules, and outputs the resulting `Measurement` object in a JSON representation again.

Functions

perform_combination(data, labels, datatype)	Concatenates segments according to *labels*.

ep_bolfi.kadi_tools.combine_measurement_segments.perform_combination

ep_bolfi.kadi_tools.combine_measurement_segments.**perform_combination**(*data*, *labels*, *datatype*, *adapt_indices=False*)

Concatenates segments according to *labels*.

Parameters

- **data** – An object from `ep_bolfi.utility.dataset_formatting` derived from `Measurement`.
- **labels** – A list of labels, corresponding to the segments in *data*. If identical labels are detected in neighbouring segments, they get concatenated, decreasing segment count by one.
- **datatype** – One of 'cycling', 'static', or 'impedance', relating to the classes defined in *dataset_formatting*.

- **adapt_indices** – Set to True if indices are numerical and you wish to keep their spacing the same as before.

Returns
The *data* object, with segments concatenated per *labels*.

ep_bolfi.kadi_tools.convert_csv_to_parquet

Stores CSV file contents in an Apache Parquet file. This optimally compresses the data and makes it faster to read.

ep_bolfi.kadi_tools.eis_parameterization

Reads in an input file prepared by `ep_bolfi.kadi_tools.eis_preprocessing` and runs EP-BOLFI on it.

ep_bolfi.kadi_tools.eis_preprocessing

Reads in an `Impedance_Measurement` json representation and preprocesses it as EIS data, assuming each segment corresponds to an equilibrium.

Functions

eis_feature_visualizer(dataset, ...)	Visualizes DRT features for each EIS segment.
eis_features(dataset, number_of_peaks, ...)	Collects DRT features from all segments into a contiguos array.
eis_features_by_segment(dataset, ...)	Calculates DRT features for each EIS segment.

ep_bolfi.kadi_tools.eis_preprocessing.eis_feature_visualizer

ep_bolfi.kadi_tools.eis_preprocessing.**eis_feature_visualizer**(*dataset*, *number_of_peaks*, *lambda_values*)

Visualizes DRT features for each EIS segment.

Parameters

- **dataset** – An `Impedance_Measurement` object.
- **number_of_peaks** – DRT peaks get detected automatically. Default is to keep all. Set to a negative integer to get the first `-number_of_peaks`, and to a positive one for the last `number_of_peaks` peaks.
- **lambda_values** – Please refer to the `pyimpspec` documentation for these. Either set one value for all segments, or a list of these for each segment. Defaults to an optimally calculated lambda value.

Returns
A list of 3-tuples, where each of them represents one semicircle from the DRT approximation. First entry is the frequencies over which the semicircle in the second entry is plotted, and the third is a label describing the semicircle parameters.

ep_bolfi.kadi_tools.eis_preprocessing.eis_features

ep_bolfi.kadi_tools.eis_preprocessing.**eis_features**(*dataset*, *number_of_peaks*, *lambda_values*)

Collects DRT features from all segments into a contiguos array.

Parameters

- **dataset** – An Impedance_Measurement object.
- **number_of_peaks** – DRT peaks get detected automatically. Default is to keep all. Set to a negative integer to get the first -number_of_peaks, and to a positive one for the last number_of_peaks peaks.
- **lambda_values** – Please refer to the pyimpspec documentation for these. Either set one value for all segments, or a list of these for each segment. Defaults to an optimally calculated lambda value.

Returns

A list that is the concatenation of all DRT features.

ep_bolfi.kadi_tools.eis_preprocessing.eis_features_by_segment

ep_bolfi.kadi_tools.eis_preprocessing.**eis_features_by_segment**(*dataset*, *number_of_peaks*, *lambda_values*)

Calculates DRT features for each EIS segment.

Parameters

- **dataset** – An Impedance_Measurement object.
- **number_of_peaks** – DRT peaks get detected automatically. Default is to keep all. Set to a negative integer to get the first -number_of_peaks, and to a positive one for the last number_of_peaks peaks.
- **lambda_values** – Please refer to the pyimpspec documentation for these. Either set one value for all segments, or a list of these for each segment. Defaults to an optimally calculated lambda value.

Returns

A 2-tuple, with the list of DRT features for each segment, and the list of complete DRT result objects themselves.

ep_bolfi.kadi_tools.eis_visualization

Reads in an input file prepared by ep_bolfi.kadi_tools.eis_preprocessing and runs and compares the simulations for the initial parameter guesses against the data.

ep_bolfi.kadi_tools.extract_measurement_protocol

Reads in a `Cycling_Information` json representation and outputs a text-based representation of the different segments that corresponds to `pybamm.Experiment` inputs.

ep_bolfi.kadi_tools.extract_ocv_curve

Reads in a `Cycling_Information` json representation and extracts OCV information from it, assuming it represents a GITT measurement.

ep_bolfi.kadi_tools.extract_overpotential

Reads in a `Cycling_Information` json representation and applies a PyBaMM model parameter set to subtract the OCP curve(s) from it, leaving us with only the overpotential.

Functions

ocv_mismatch(soc, ocv, parameters[, ...])	Compares data OCV with model OCV(SOC) to determine their mismatch.
transform_to_unity_interval(segment)	Transforms the *segment* list to start at 0 and end at 1.

ep_bolfi.kadi_tools.extract_overpotential.ocv_mismatch

ep_bolfi.kadi_tools.extract_overpotential.**ocv_mismatch**(*soc*, *ocv*, *parameters*, *electrode='both'*, *voltage_sign=0*)

Compares data OCV with model OCV(SOC) to determine their mismatch.

Parameters

- **soc** – The SOC data points.
- **ocv** – The OCV data points.
- **parameters** – The PyBaMM model parameters containing the model OCP curves.
- **electrode** – Set to 'positive' or 'negative' for three-electrode setups or half-cell setups, or 'both' to subtract both electrode OCPs.
- **voltage_sign** – Optional prefactor to multiply the model OCV with before subtracting it from the data OCV when `electrode != 'both'`. Has no effect otherwise. Defaults to usual sign conventions.

Returns

The difference between data OCV and model OCV(SOC).

ep_bolfi.kadi_tools.extract_overpotential.transform_to_unity_interval

ep_bolfi.kadi_tools.extract_overpotential.**transform_to_unity_interval**(*segment*)

Transforms the *segment* list to start at 0 and end at 1.

Parameters
segment – A list of numbers.

Returns
A linearly transformed *segment*, so that it lives on [0,1].

ep_bolfi.kadi_tools.fit_and_plot_ocv

Reads in a json file containing OCV information and fits the model from Birkl2015 (A Parametric OCV Model for Li-Ion Batteries, DOI 10.1149/2.0331512jes) to it.

ep_bolfi.kadi_tools.fit_exponential_decay

Reads in a `Cycling_Information` json representation and fits an exponential fit function to its first segment.

ep_bolfi.kadi_tools.gitt_parameterization

Reads in a `Cycling_Information` json representation and uses it in conjunction with input files generated by `ep_bolfi.kadi_tools.gitt_preprocessing` to fit any PyBaMM model to it via EP-BOLFI.

ep_bolfi.kadi_tools.gitt_preprocessing

Reads in a `Cycling_Information` json representation and preprocesses it as GITT data, assuming each segment corresponds to a pulse or pause. Additionally formats all necessary inputs for EP-BOLFI parameterization.

Functions

gitt_feature_visualizer(dataset, ...)	Visualizes diffusion features for each GITT segment.
gitt_features(dataset, ...)	Calculates diffusion features for each GITT segment. They are flattened into one list, with each five entries referring to: - asymptotic concentration overpotential - exponential magnitude - exponential timescale - initial voltage - inverse square-root slope at beginning (d√t / dU).

ep_bolfi.kadi_tools.gitt_preprocessing.gitt_feature_visualizer

ep_bolfi.kadi_tools.gitt_preprocessing.**gitt_feature_visualizer**(*dataset*, *list_of_feature_indices*, *sqrt_cutoff*, *sqrt_start*, *exp_cutoff*)

Visualizes diffusion features for each GITT segment.

Parameters

- **dataset** – A `Cycling_Information` object.
- **list_of_feature_indices** – The indices from the whole list to retain for fitting.
- **sqrt_cutoff** – The upper time limit on data to use in each segment for the square-root fit. The time limit counts from the segment start.
- **sqrt_start** – The lower time limit on data to use in each segment for the square-root fit. The time limit counts from the segment start.
- **exp_cutoff** – The lower time limit on data to use in each segment for the exponential fit. The time limit counts from the segment end.

Returns

A list of 3-tuples, where each of them represents a square-root or exponential fit to a segment. First entry is the times over which the voltage in the second entry is plotted, and the third is a label describing the square-root or exponential parameters.

ep_bolfi.kadi_tools.gitt_preprocessing.gitt_features

ep_bolfi.kadi_tools.gitt_preprocessing.**gitt_features**(*dataset*, *list_of_feature_indices*, *sqrt_cutoff*, *sqrt_start*, *exp_cutoff*)

Calculates diffusion features for each GITT segment. They are flattened into one list, with each five entries referring to:

- asymptotic concentration overpotential
- exponential magnitude
- exponential timescale
- initial voltage
- inverse square-root slope at beginning (d√t / dU)

Parameters

- **dataset** – A `Cycling_Information` object.
- **list_of_feature_indices** – The indices from the whole list to retain for fitting.
- **sqrt_cutoff** – The upper time limit on data to use in each segment for the square-root fit. The time limit counts from the segment start.
- **sqrt_start** – The lower time limit on data to use in each segment for the square-root fit. The time limit counts from the segment start.
- **exp_cutoff** – The lower time limit on data to use in each segment for the exponential fit. The time limit counts from the segment end.

Returns
The flattened list of selected diffusion features.

ep_bolfi.kadi_tools.gitt_visualization

Reads in an input file prepared by `ep_bolfi.kadi_tools.gitt_preprocessing` and runs and compares the simulations for the initial parameter guesses against the data.

Functions

plot_voltage_components(input_data, parameters)	Generate a plot showing the component overpotentials that make up the voltage

ep_bolfi.kadi_tools.gitt_visualization.plot_voltage_components

ep_bolfi.kadi_tools.gitt_visualization.**plot_voltage_components**(*input_data*, *parameters*, *ax=None*, *show_legend=True*, *split_by_electrode=False*, *three_electrode=None*, *dimensionless_reference_electrode_location=0.5*, *show_plot=True*, ***kwargs_fill*)

Generate a plot showing the component overpotentials that make up the voltage

Parameters

input_data
[`pybamm.Solution` or `pybamm.Simulation`] Solution or Simulation object from which to extract voltage components.

ax
[matplotlib Axis, optional] The axis on which to put the plot. If None, a new figure and axis is created.

show_legend
[bool, optional] Whether to display the legend. Default is True

split_by_electrode
[bool, optional] Whether to show the overpotentials for the negative and positive electrodes separately. Default is False.

three_electrode
[str, optional] With None, does nothing (i.e., cell potentials are used). If set to either 'positive' or 'negative', instead of cell potentials, the base for the displayed voltage will be the potential of the 'positive' or 'negative' electrode against a reference electrode. For placement of said reference electrode, please refer to "dimensionless_reference_electrode_location".

dimensionless_reference_electrode_location
[float, optional] The location of the reference electrode, given as a scalar between 0 (placed at the point where negative electrode and separator meet) and 1 (placed at the point where positive electrode and separator meet). Defaults to 0.5 (in the middle).

show_plot
[bool, optional] Whether to show the plots. Default is True. Set to False if you want to only display the plot after plt.show() has been called.

kwargs_fill
Keyword arguments: `matplotlib.axes.Axes.fill_between`

ep_bolfi.kadi_tools.plot_measurement

Reads in a `Cycling_Information` json representation and plots it.

ep_bolfi.kadi_tools.read_csv_datasets

Preprocesses commonly encountered CSV measurement files to the datatypes defined in `ep_bolfi.utility.dataset_formatting`.

ep_bolfi.kadi_tools.read_hdf5_datasets

Provides preprocessing from HDF5 contents to the datatypes defined in `ep_bolfi.utility.dataset_formatting`.

ep_bolfi.kadi_tools.read_measurement_from_parquet

Reads a `Measurement` object that got stored in an Apache Parquet file with `ep_bolfi.kadi_tools.store_measurement_in_parquet`.

ep_bolfi.kadi_tools.select_measurement_segments

Selects specified segments of the `Measurement` and returns them.

ep_bolfi.kadi_tools.store_measurement_as_parquet

Stores a `Measurement` object in an Apache Parquet file. This optimally compresses the data and makes it faster to read.

2.1.2 ep_bolfi.models

Modules

analytic_impedance	Contains equations to evaluate the analytic impedance of the SPMe model.
assess_effective_parameters	Evaluates a parameter set for, e.g., overpotential and capacity.
electrolyte	Contains a PyBaMM-compatible electrolyte model.
equivalent_circuits	
solversetup	This file eases the setup and simulation of PyBaMM battery models.
standard_parameters	Comprehensive list of parameters of every model.

ep_bolfi.models.analytic_impedance

Contains equations to evaluate the analytic impedance of the SPMe model.

Classes

AnalyticImpedance(parameters[, ...])	Analytic impedance of the SPMe.

ep_bolfi.models.analytic_impedance.AnalyticImpedance

class ep_bolfi.models.analytic_impedance.**AnalyticImpedance**(*parameters*, *catch_warnings=True*, *precision=106*, *verbose=False*)

Bases: `object`

Analytic impedance of the SPMe.

__init__(*parameters*, *catch_warnings=True*, *precision=106*, *verbose=False*)

Preprocesses the model parameters.

Parameters

- **parameters** – A dictionary of parameter values with the parameter names as keys. For these names, please refer to models.standard_parameters.
- **catch_warnings** – Set to False if you want to disable the over-/underflow protection.
- **precision** – Precision (in bits) in which the computations are carried out. Note that the library used for this (mpmath) sets this globally. The default is double of what a 64-bit float usually assigns to the decimal number (53 bits). Exponents are stored exactly.
- **verbose** – If set to True, information about the so-called resonancy frequencies and resistances will be printed during model calculations. The resonancy frequencies are the frequencies with maximum imaginary part of the impedance for the various model components. The resistances are the widths of the semi-circles or Warburg two-thirds-circles with 45° line.

Methods

An(s_nondim)	Integration constant; please refer to the PDF.
As(s_nondim)	Integration constant; please refer to the PDF.
As_metal_counter(s_nondim)	Integration constant; please refer to the PDF.
Bs(s_nondim)	Integration constant; please refer to the PDF.
Bs_metal_counter(s_nondim)	Integration constant; please refer to the PDF.
Cp(s_nondim)	Integration constant; please refer to the PDF.
Cp_metal_counter(s_nondim)	Integration constant; please refer to the PDF.
Z_DL(s_dim[, working_electrode])	Impedance of the Double-Layers.
Z_SEI(s_dim)	Impedance of the Double-Layer.
Z_SPM(s_dim)	Transfer function U(s) / I(s) of the SPM.
Z_SPM_metal_counter(s_dim)	Transfer function U(s) / I(s) of the SPM (metal CE).
Z_SPM_offset()	Static part of the transfer function of the SPM.
Z_SPM_offset_metal_counter()	Static part of the transfer function of the SPM (metal CE).
Z_SPM_offset_reference_electrode([...])	Impedance against a reference electrode.
Z_SPM_reference_electrode(s_dim[, ...])	Impedance against a reference electrode.
Z_SPMe(s_dim)	Transfer function U(s) / I(s) of the SPMe.
Z_SPMe_1(s_dim)	Additional correction to the SPM's transfer function.
Z_SPMe_1_metal_counter(s_dim)	Correction to the transfer function of the SPM (metal CE).
Z_SPMe_1_offset()	Static part of the correction to the SPM's impedance.
Z_SPMe_1_offset_metal_counter()	
Z_SPMe_1_offset_reference_electrode([...])	Impedance against a reference electrode.
Z_SPMe_1_reference_electrode(s_dim[, ...])	Impedance against a reference electrode.
Z_SPMe_metal_counter(s_dim)	Transfer function U(s) / I(s) of the SPMe (metal CE).
Z_SPMe_offset()	Static part of the transfer function of the SPMe.
Z_SPMe_offset_metal_counter()	Static part of the half-cell SPMe's transfer function.
Z_SPMe_offset_reference_electrode([...])	Impedance against a reference electrode.
Z_SPMe_reference_electrode(s_dim[, ...])	Impedance against a reference electrode.
Z_SPMe_with_double_layer_and_SEI(s_dim)	Impedance of the SPMe/DFN with Double-Layer and SEI.
Z_SPMe_with_double_layer_and_SEI_offset()	Impedance of the SPMe/DFN with Double-Layer and SEI.
Z_SPMe_with_double_layer_and_SEI_reference_electrode(s_dim)	Impedance against a reference electrode.
__init__(parameters[, catch_warnings, ...])	Preprocesses the model parameters.
bar_cen_1(s_nondim)	Integration constant; please refer to the PDF.
bar_cep_1(s_nondim)	Integration constant; please refer to the PDF.
bar_cep_1_metal_counter(s_nondim)	Integration constant; please refer to the PDF.
cen_1(s_nondim, x_nondim)	Electrolyte concentration within the neg.
cep_1(s_nondim, x_nondim)	Electrolyte concentration within the pos.
cep_1_metal_counter(s_nondim, x_nondim)	Electrolyte concentration against Li metal (pos.
ces_1(s_nondim, x_nondim)	Electrolyte concentration within the separator.
ces_1_metal_counter(s_nondim, x_nondim)	Electrolyte concentration against Li metal (separator).
d_dx_cen_1(s_nondim, x_nondim)	Electrolyte concentration gradient (neg.

continues on next page

Table 1 – continued from previous page

d_dx_cep_1(s_nondim, x_nondim)	Electrolyte concentration gradient (pos.
d_dx_cep_1_metal_counter(s_nondim, x_nondim)	Electrolyte gradient against Li metal (pos.
d_dx_ces_1(s_nondim, x_nondim)	Electrolyte concentration within the separator.
d_dx_ces_1_metal_counter(s_nondim, x_nondim)	Electrolyte gradient against Li metal (separator).

Attributes

parameters	The parameter dictionary converted to PyBaMM form.
symbolic_constants	All relevant constants in a dictionary.
constants	self.constants["c_{e0}^1 / I"] = (
Rp_int	Positive electrode intercalation resistance.
Zp_int	Positive electrode intercalation impedance scaling.
Rn_int	Negative electrode intercalation resistance.
Zn_int	Negative electrode intercalation impedance scaling.
Z_cep	Positive electrolyte concentration impedance scaling
Z_ces	Separator electrolyte concentration impedance scaling.
Z_cen	Negative electrolyte concentration impedance scaling.
Rsen	Negative electrode charge transfer resistance.
Rsep	Positive electrode charge transfer resistance.
Rse	Charge transfer resistance.
Re	Electrolyte conductivity resistance.
Re_metal_counter	Electrolyte conductivity resistance (metal counter electrode).
Rs	Electrode conductivity resistance.
Rsn	Negative electrode conductivity resistance.
Rsp	Positive electrode conductivity resistance.
Rs_metal_counter	Electrode conductivity resistance (metal counter electrode).
κe	Ionic conductivity of the electrolyte.
timescale	Timescale used for non-dimensionalization in s.
thermal_voltage	Voltage used for non-dimensionalization in V.
C	C-rate of the battery in A.
Ce	Non-dimensionalized timescale of the electrolyte.
De	Non-dimensionalized diffusivity of the electrolyte.
De_dim	Diffusivity of the electrolyte.
βp	Bruggeman coefficient of the cathode.
βs	Bruggeman coefficient of the separator.
βn	Bruggeman coefficient of the anode.
εp	Porosity of the cathode.
εs	Porosity of the separator.
εn	Porosity of the anode.
Lp	Fraction of the cathode length of the battery length.
Ls	Fraction of the separator length of the battery length.
Ln	Fraction of the anode length of the battery length.
t_plus	Transference number.
ye	Non-dimensionalized electrolyte concentration.
one_plus_dlnf_dlnc	Thermodynamic factor.
zp	Charge number for the cathode.

continues on next page

Table 2 – continued from previous page

zn	Charge number for the anode.
C_DLn	Negative electrode double-layer capacity.
C_DLp	Positive electrode double-layer capacity.
L_SEI	Thickness of the SEI.
R_SEI	Resistance of the SEI.
C_SEI	Capacitance of the SEI.
np	Stoichiometry of salt dissociation (cation).
nn	Stoichiometry of salt dissociation (anion).
zp_salt	Charge number of salt dissociation (cation).
zn_salt	Charge number of salt dissociation (anion).
ρe	Electrolyte mass density.
ρe_plus	Cation mass density in electrolyte.
M_N	Molar mass of electrolyte solvent.
c_N	Solvent concentration.
ρ_N	Solvent mass density.
v_N	Solvent partial molar volume.
tilde_ρ_N	Solvent partial mass density.
ε_SEI	SEI porosity.
β_SEI	SEI Bruggeman coefficient (for tortuosity).
L_electrolyte_for_SEI_model	Length-scale for bulk electrolyte contribution in SEI model.
t_SEI_minus	Anion transference number in SEI.
F	Faraday constant.

An (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the SPMe. Note: compared to its formulation in the appended PDF, they have already been scaled with I(s).

Parameters
s_nondim – The non-dimensionalized frequencies as an array.

Returns
Integration constant of the solution of c_e^1 in the SPMe.

As (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the SPMe. Note: compared to its formulation in the appended PDF, they have already been scaled with I(s).

Parameters
s_nondim – The non-dimensionalized frequencies as an array.

Returns
Integration constant of the solution of c_e^1 in the SPMe.

As_metal_counter (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the metal CE SPMe. Note: compared to its formulation in the appended PDF, they have already been scaled with I(s).

Parameters
s_nondim – The non-dimensionalized frequencies as an array.

Returns
Integration constant of the solution of c_e^1 in the half-cell SPMe.

Bs (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the SPMe. Note: compared to its formulation in the appended PDF, they have already been scaled with I(s).

Parameters
s_nondim – The non-dimensionalized frequencies as an array.

Returns
Integration constant of the solution of c_e^1 in the SPMe.

Bs_metal_counter (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the metal CE SPMe. Note: compared to its formulation in the appended PDF, they have already been scaled with I(s).

Parameters
s_nondim – The non-dimensionalized frequencies as an array.

Returns
Integration constant of the solution of c_e^1 in the half-cell SPMe.

C

C-rate of the battery in A.

C_DLn

Negative electrode double-layer capacity.

C_DLp

Positive electrode double-layer capacity.

C_SEI

Capacitance of the SEI.

Ce

Non-dimensionalized timescale of the electrolyte.

Cp (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the SPMe. Note: compared to its formulation in the appended PDF, C□(s) here is A□(s) exp(Z_c□ sqrt(s)) in the PDF.

Parameters
s – The non-dimensionalized frequencies as an array.

Returns
Integration constant of the solution of c_e^1 in the SPMe.

Cp_metal_counter (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the metal CE SPMe

Parameters
s_nondim – The non-dimensionalized frequencies as an array.

Returns

Integration constant of the solution of c_e^1 in the half-cell SPMe.

De

Non-dimensionalized diffusivity of the electrolyte.

De_dim

Diffusivity of the electrolyte.

F

Faraday constant.

L_SEI

Thickness of the SEI.

L_electrolyte_for_SEI_model

Length-scale for bulk electrolyte contribution in SEI model.

Ln

Fraction of the anode length of the battery length.

Lp

Fraction of the cathode length of the battery length.

Ls

Fraction of the separator length of the battery length.

M_N

Molar mass of electrolyte solvent.

R_SEI

Resistance of the SEI.

Re

Electrolyte conductivity resistance.

Re_metal_counter

Electrolyte conductivity resistance (metal counter electrode).

Rn_int

Negative electrode intercalation resistance.

Rp_int

Positive electrode intercalation resistance.

Rs

Electrode conductivity resistance.

Rs_metal_counter

Electrode conductivity resistance (metal counter electrode).

Rse

Charge transfer resistance.

Rsen

Negative electrode charge transfer resistance.

Rsep

Positive electrode charge transfer resistance.

Rsn

Negative electrode conductivity resistance.

Rsp

Positive electrode conductivity resistance.

Z_DL(*s_dim*, *working_electrode=None*)

Impedance of the Double-Layers.

Parameters

- **s_dim** – An array of the frequencies to evaluate.
- **working_electrode** – The electrode against which the voltage was measured with respect to the reference electrode. 'positive' or 'negative'. Defaults to both electrodes contributing to the impedance.

Returns

The evaluated impedances as an array.

Z_SEI(*s_dim*)

Impedance of the Double-Layer.

Parameters

s_dim – An array of the frequencies to evaluate.

Returns

The evaluated impedances as an array.

Z_SPM(*s_dim*)

Transfer function U(s) / I(s) of the SPM.

Parameters

s_dim – An array of the frequencies to evaluate.

Returns

The evaluated impedances as an array.

Z_SPM_metal_counter(*s_dim*)

Transfer function U(s) / I(s) of the SPM (metal CE).

Parameters

s_dim – An array of the frequencies to evaluate.

Returns

The evaluated impedances as an array.

Z_SPM_offset()

Static part of the transfer function of the SPM.

Returns

The part of the SPM's impedance that doesn't depend on frequency.

Z_SPM_offset_metal_counter()

Static part of the transfer function of the SPM (metal CE).

Returns

The part of the half-cell SPM's impedance that doesn't depend on frequency.

Z_SPM_offset_reference_electrode(*working_electrode='positive'*)

Impedance against a reference electrode.

Parameters

- **s_dim** – An array of the frequencies to evaluate.
- **working_electrode** – The electrode against which the voltage was measured with respect to the reference electrode. 'positive' or 'negative'.

Returns

The evaluated impedances as an array.

Z_SPM_reference_electrode(*s_dim*, *working_electrode='positive'*)

Impedance against a reference electrode.

Parameters

- **s_dim** – An array of the frequencies to evaluate.
- **working_electrode** – The electrode against which the voltage was measured with respect to the reference electrode. 'positive' or 'negative'.

Returns

The evaluated impedances as an array.

Z_SPMe(*s_dim*)

Transfer function U(s) / I(s) of the SPMe.

Parameters

s_dim – An array of the frequencies to evaluate.

Returns

The evaluated impedances as an array.

Z_SPMe_1(*s_dim*)

Additional correction to the SPM's transfer function.

Parameters

s_dim – An array of the frequencies to evaluate.

Returns

The evaluated impedances as an array.

Z_SPMe_1_metal_counter(*s_dim*)

Correction to the transfer function of the SPM (metal CE).

Parameters

s_dim – An array of the frequencies to evaluate.

Returns

The evaluated impedances as an array.

Z_SPMe_1_offset()

Static part of the correction to the SPM's impedance.

Returns

The part of the difference between the SPM's and SPMe's impedance that doesn't depend on frequency.

Z_SPMe_1_offset_metal_counter()

Static part of the impedance of the SPMe(1) (metal CE).

Returns
The part of the difference between the SPM's and SPMe's impedance for half-cells that doesn't depend on frequency.

Z_SPMe_1_offset_reference_electrode(*working_electrode='positive'*, *dimensionless_location=0.5*)

Impedance against a reference electrode.

Parameters

- **s_dim** – An array of the frequencies to evaluate.
- **working_electrode** – The electrode against which the voltage was measured with respect to the reference electrode. 'positive' or 'negative'.
- **dimensionless_location** – The location of the reference electrode. 0 refers to the point where negative electrode and separator touch, and 1 refers to the point where separator and positive electrode touch.

Returns
The evaluated impedances as an array.

Z_SPMe_1_reference_electrode(*s_dim*, *working_electrode='positive'*, *dimensionless_location=0.5*)

Impedance against a reference electrode.

Parameters

- **s_dim** – An array of the frequencies to evaluate.
- **working_electrode** – The electrode against which the voltage was measured with respect to the reference electrode. 'positive' or 'negative'.
- **dimensionless_location** – The location of the reference electrode. 0 refers to the point where negative electrode and separator touch, and 1 refers to the point where separator and positive electrode touch.

Returns
The evaluated impedances as an array.

Z_SPMe_metal_counter(*s_dim*)

Transfer function U(s) / I(s) of the SPMe (metal CE).

Parameters
s_dim – An array of the frequencies to evaluate.

Returns
The evaluated impedances as an array.

Z_SPMe_offset()

Static part of the transfer function of the SPMe.

Returns
The part of the SPMe's impedance that doesn't depend on frequency.

Z_SPMe_offset_metal_counter()

Static part of the half-cell SPMe's transfer function.

Returns
The part of the half-cell SPMe's impedance that doesn't depend on frequency.

Z_SPMe_offset_reference_electrode(*working_electrode='positive'*)

Impedance against a reference electrode.

Parameters
working_electrode – The electrode against which the voltage was measured with respect to the reference electrode. 'positive' or 'negative'.

Returns
The evaluated impedances as an array.

Z_SPMe_reference_electrode(*s_dim*, *working_electrode='positive'*, *dimensionless_location=0.5*)

Impedance against a reference electrode.

Parameters

- **s_dim** – An array of the frequencies to evaluate.
- **working_electrode** – The electrode against which the voltage was measured with respect to the reference electrode. 'positive' or 'negative'.
- **dimensionless_location** – The location of the reference electrode. 0 refers to the point where negative electrode and separator touch, and 1 refers to the point where separator and positive electrode touch.

Returns
The evaluated impedances as an array.

Z_SPMe_with_double_layer_and_SEI(*s_dim*)

Impedance of the SPMe/DFN with Double-Layer and SEI.

Parameters
s_dim – An array of the frequencies to evaluate.

Returns
The evaluated impedances as an array.

Z_SPMe_with_double_layer_and_SEI_offset()

Impedance of the SPMe/DFN with Double-Layer and SEI.

Returns
The evaluated impedances as an array.

Z_SPMe_with_double_layer_and_SEI_reference_electrode(*s_dim*, *working_electrode='positive'*, *dimensionless_location=0.5*)

Impedance against a reference electrode.

Parameters

- **s_dim** – An array of the frequencies to evaluate.
- **working_electrode** – The electrode against which the voltage was measured with respect to the reference electrode. 'positive' or 'negative'.
- **dimensionless_location** – The location of the reference electrode. 0 refers to the point where negative electrode and separator touch, and 1 refers to the point where separator and positive electrode touch.

Returns
The evaluated impedances as an array.

Z_cen

Negative electrolyte concentration impedance scaling.

Z_cep

Positive electrolyte concentration impedance scaling

Z_ces

Separator electrolyte concentration impedance scaling.

Zn_int

Negative electrode intercalation impedance scaling.

Zp_int

Positive electrode intercalation impedance scaling.

bar_cen_1 (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the full-cell SPMe. Note: compared to its formulation in the appended PDF, it has already been scaled with I(s).

Parameters

s_nondim – The non-dimensionalized frequencies as an array.

Returns

Integration constant of the solution of c_e^1 in the half-cell SPMe.

bar_cep_1 (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the full-cell SPMe. Note: compared to its formulation in the appended PDF, it has already been scaled with I(s).

Parameters

s_nondim – The non-dimensionalized frequencies as an array.

Returns

Integration constant of the solution of c_e^1 in the half-cell SPMe.

bar_cep_1_metal_counter (*s_nondim*)

Integration constant; please refer to the PDF.

Integration constant of the solution of c_e^1 in the metal CE SPMe. Note: compared to its formulation in the appended PDF, they have already been scaled with I(s).

Parameters

s_nondim – The non-dimensionalized frequencies as an array.

Returns

Integration constant of the solution of c_e^1 in the half-cell SPMe.

c_N

Solvent concentration.

cen_1 (*s_nondim*, *x_nondim*)

Electrolyte concentration within the neg. electrode.

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.

- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the neg. electrode.

Returns
The electrolyte concentration.

cep_1 (*s_nondim*, *x_nondim*)

Electrolyte concentration within the pos. electrode.

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the pos. electrode.

Returns
The electrolyte concentration.

cep_1_metal_counter (*s_nondim*, *x_nondim*)

Electrolyte concentration against Li metal (pos. electrode).

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the pos. electrode.

Returns
The electrolyte concentration.

ces_1 (*s_nondim*, *x_nondim*)

Electrolyte concentration within the separator.

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the separator.

Returns
The electrolyte concentration.

ces_1_metal_counter (*s_nondim*, *x_nondim*)

Electrolyte concentration against Li metal (separator).

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the separator.

Returns
The electrolyte concentration.

constants

self.constants["c_{e0}^1 / I"] = (
(1 - t_plus(c_e_init)) / (6 * D_e(c_e_init) * γ_e) * (2 * L□**2 / ε□_scalar**β_e□ - 2 * L□**2 / ε□_scalar**β_e□ + 3 / ε□_scalar**β_e□ * (L□**2 - L□**2 + 1)))

self.constants.update({

"c_e[illegible]^1 / I": self.constants["c_{e0}^1 / I"]

- (1 - t_plus(c_e_init)) / (6 * D_e(c_e_init) * γ_e)
- (2 * L[illegible] / ε[illegible]_scalar**β_e[illegible] - 6 * L[illegible] / ε[illegible]_scalar**β_e[illegible]),

"c_e[illegible]^1 / I": self.constants["c_{e0}^1 / I"]

- (1 - t_plus(c_e_init)) / (6 * D_e(c_e_init) * γ_e)
- (-2 * L[illegible] / ε[illegible]_scalar**β_e[illegible] - 6 * (1 - L[illegible])/ε[illegible]_scalar**β_e[illegible]),

})

d_dx_cen_1 (*s_nondim*, *x_nondim*)

Electrolyte concentration gradient (neg. electrode).

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the neg. electrode.

Returns

The electrolyte concentration gradient.

d_dx_cep_1 (*s_nondim*, *x_nondim*)

Electrolyte concentration gradient (pos. electrode).

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the pos. electrode.

Returns

The electrolyte concentration gradient.

d_dx_cep_1_metal_counter (*s_nondim*, *x_nondim*)

Electrolyte gradient against Li metal (pos. electrode).

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the pos. electrode.

Returns

The electrolyte concentration gradient.

d_dx_ces_1 (*s_nondim*, *x_nondim*)

Electrolyte concentration within the separator.

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the separator.

Returns
The electrolyte concentration.

d_dx_ces_1_metal_counter(*s_nondim*, *x_nondim*)

Electrolyte gradient against Li metal (separator).

Parameters

- **s_nondim** – The non-dimensionalized frequencies as an array.
- **x_nondim** – The non-dimensionalized location within the cell. 0 is the negative current collector and 1 the positive one. Return values are only valid within the separator.

Returns
The electrolyte concentration gradient.

nn

Stoichiometry of salt dissociation (anion).

np

Stoichiometry of salt dissociation (cation).

one_plus_dlnf_dlnc

Thermodynamic factor.

parameters

The parameter dictionary converted to PyBaMM form.

symbolic_constants

All relevant constants in a dictionary. Note: any term here that corresponds to U^1 - η^1 + η^1 is already scaled by C_e / I. All constants herein are duplicated as members after this. Their descriptions match their names (the keys of this dictionary).

t_SEI_minus

Anion transference number in SEI.

t_plus

Transference number.

thermal_voltage

Voltage used for non-dimensionalization in V.

tilde_ρ_N

Solvent partial mass density.

timescale

Timescale used for non-dimensionalization in s.

v_N

Solvent partial molar volume.

zn

Charge number for the anode.

zn_salt

Charge number of salt dissociation (anion).

zp

Charge number for the cathode.

zp_salt
Charge number of salt dissociation (cation).

β_SEI
SEI Bruggeman coefficient (for tortuosity).

βn
Bruggeman coefficient of the anode.

βp
Bruggeman coefficient of the cathode.

βs
Bruggeman coefficient of the separator.

γe
Non-dimensionalized electrolyte concentration.

ε_SEI
SEI porosity.

εn
Porosity of the anode.

εp
Porosity of the cathode.

εs
Porosity of the separator.

κe
Ionic conductivity of the electrolyte.

ρ_N
Solvent mass density.

ρe
Electrolyte mass density.

ρe_plus
Cation mass density in electrolyte.

ep_bolfi.models.assess_effective_parameters

Evaluates a parameter set for, e.g., overpotential and capacity.

Classes

Effective_Parameters(parameters[, ...])	Calculates, stores and prints effective parameters.

ep_bolfi.models.assess_effective_parameters.Effective_Parameters

class ep_bolfi.models.assess_effective_parameters.**Effective_Parameters**(*parameters*, *working_electrode='both'*)

Bases: `object`

Calculates, stores and prints effective parameters.

__init__(*parameters*, *working_electrode='both'*)

Preprocesses the model parameters.

Parameters

- **parameters** – A dictionary of parameter values with the parameter names as keys. For these names, please refer to ep_bolfi.models.standard_parameters.
- **working_electrode** – When set to either 'negative' or 'positive', the parameters will be treated as a half-cell setup with said electrode.

Methods

__init__(parameters[, working_electrode])	Preprocesses the model parameters.
eval(expression)	Short-hand for PyBaMM symbol evaluation.
print([c_rates])	Prints the voltage losses for the given C-rates.

Attributes

parameters	The parameter dictionary converted to PyBaMM form.
working_electrode	Sets whether full- or half-cell parameters are calculated.
Qn	Negative electrode theoretical capacity.
Qp	Positive electrode theoretical capacity.
R_ce_0_1	Integration constant for the electrolyte concentration / I.
R_bar_cen_1	Electrolyte concentration / I at the negative electrode.
R_bar_cep_1	Electrolyte concentration / I at the positive electrode.
R_bar_φsn_1	Effective resistance of the negative electrode.
R_bar_φsp_1	Effective resistance of the positive electrode.
Rns	Negative electrode resistance (as calculated by the SPMe(S)).
Rps	Positive electrode resistance (as calculated by the SPMe(S)).
Re	Electrolyte resistance (as calculated by the SPMe(S)).
isen	SPM(e) negative electrode exchange-current.
isep	SPM(e) positive electrode exchange-current.

Qn

Negative electrode theoretical capacity.

Qp

Positive electrode theoretical capacity.

R_bar_cen_1

Electrolyte concentration / I at the negative electrode.

R_bar_cep_1

Electrolyte concentration / I at the positive electrode.

R_bar_φsn_1

Effective resistance of the negative electrode.

R_bar_φsp_1

Effective resistance of the positive electrode.

R_ce_0_1

Integration constant for the electrolyte concentration / I.

Re

Electrolyte resistance (as calculated by the SPMe(S)).

Rns

Negative electrode resistance (as calculated by the SPMe(S)).

Rps

Positive electrode resistance (as calculated by the SPMe(S)).

eval(*expression*)

Short-hand for PyBaMM symbol evaluation.

Parameters
expression – A pybamm.Symbol.

Returns
The numeric value of "expression".

isen

SPM(e) negative electrode exchange-current.

isep

SPM(e) positive electrode exchange-current.

parameters

The parameter dictionary converted to PyBaMM form. Convert SubstitutionDict to dict by iterating over it.

print(*c_rates=[0.1, 0.2, 0.5, 1.0]*)

Prints the voltage losses for the given C-rates.

Parameters
c_rates – The C-rates (as fraction of "Typical current [A]").

working_electrode

Sets whether full- or half-cell parameters are calculated.

ep_bolfi.models.electrolyte

Contains a PyBaMM-compatible electrolyte model.

Classes

Electrolyte([pybamm_control, name, options, ...])	Electrolyte model assuming a symmetric Li-metal cell.
Electrolyte_internal(param[, ...])	Defining equations for a symmetric Li cell with electrolyte.

ep_bolfi.models.electrolyte.Electrolyte

class ep_bolfi.models.electrolyte.**Electrolyte**(*pybamm_control=False*, *name='electrolyte'*, *options={}*, *build=True*)

Bases: BaseBatteryModel

Electrolyte model assuming a symmetric Li-metal cell.

__init__(*pybamm_control=False*, *name='electrolyte'*, *options={}*, *build=True*)

Sets up an electrolyte model usable by PyBaMM.

Parameters

- **pybamm_control** – Per default False, which indicates that the current is given as a function. If set to True, this model is compatible with PyBaMM experiments, e.g. CC-CV simulations. The current is then a variable and it or voltage can be fixed functions.
- **name** – The optional name of the model. Default is "electrolyte".
- **options** – Used for external circuit if pybamm_control is True.
- **build** – Per default True, which builds the model equations right away. If set to False, they remain as symbols for later.

Methods

__init__([pybamm_control, name, options, build])	Sets up an electrolyte model usable by PyBaMM.
build_coupled_variables()	
build_fundamental()	
build_model()	
build_model_equations()	
check_algebraic_equations(post_discretisation)	Check that the algebraic equations are well-posed.
check_and_combine_dict(dict1, dict2)	
check_discretised_or_discretise_inplace_if_0D()	Discretise model if it isn't already discretised This only works with purely 0D models, as otherwise the mesh and spatial method should be specified by the user
check_for_time_derivatives()	
check_ics_bcs()	Check that the initial and boundary conditions are well-posed.

continues on next page

Table 3 – continued from previous page

check_no_repeated_keys()	Check that no equation keys are repeated.
check_variables()	
check_well_determined(post_discretisation)	Check that the model is not under- or over-determined.
check_well_posedness([post_discretisation])	Check that the model is well-posed by executing the following tests: - Model is not over- or underdetermined, by comparing keys and equations in rhs and algebraic.
deserialise(properties)	Create a model instance from a serialised object.
export_casadi_objects(variable_names[, ...])	Export the constituent parts of the model (rhs, algebraic, initial conditions, etc) as casadi objects.
generate(filename, variable_names[, ...])	Generate the model in C, using CasADi.
generic_deserialise(instance, properties)	
get_intercalation_kinetics(domain)	
get_inverse_intercalation_kinetics()	
get_parameter_info([by_submodel])	Extracts the parameter information and returns it as a dictionary.
info(symbol_name)	Provides helpful summary information for a symbol.
latexify([filename, newline, output_variables])	Converts all model equations in latex.
new_copy([build])	Create an empty copy with identical options.
print_parameter_info([by_submodel])	Print parameter information in a formatted table from a dictionary of parameters
process_parameters_and_discretise(symbol, ...)	Process parameters and discretise a symbol using supplied parameter values and discretisation.
save_model([filename, mesh, variables])	Write out a discretised model to a JSON file
set_current_collector_submodel()	
set_degradation_variables()	Set variables that quantify degradation.
set_external_circuit_submodel()	Define how the external circuit defines the boundary conditions for the model, e.g. (not necessarily constant-) current, voltage, etc.
set_initial_conditions_from(solution[, ...])	Update initial conditions with the final states from a Solution object or from a dictionary.
set_interface_utilisation_submodel()	
set_soc_variables()	Set variables relating to the state of charge.
set_standard_output_variables()	Adds "the horizontal axis" to self.variables.
set_summary_variables()	
set_surface_temperature_submodel()	
set_thermal_submodel()	
set_transport_efficiency_submodels()	
set_voltage_variables()	Adds voltage-specific variables to self.variables.
update(*submodels)	Update model to add new physics from submodels
variable_names()	

Attributes

algebraic	
boundary_conditions	
concatenated_algebraic	
concatenated_initial_conditions	
concatenated_rhs	
default_geometry	Override: corrects the geometry for half-cells.
default_parameter_values	
default_quick_plot_variables	
default_solver	Return default solver based on whether model is ODE/DAE or algebraic
default_spatial_methods	
default_submesh_types	
default_var_pts	
events	
geometry	
initial_conditions	
input_parameters	Returns all the input parameters in the model
jacobian	
jacobian_algebraic	
jacobian_rhs	
length_scales	Non-dimensionalization length scales.
mass_matrix	
mass_matrix_inv	
name	
options	
param	Contains all the relevant parameters for this model.
parameters	Returns all the parameters in the model
rhs	
summary_variables	
timescale	Non-dimensionalization timescale.
variables	
variables_and_events	Returns variables and events in a single dictionary
pybamm_control	Current is fixed if False and a variable if True.

check_algebraic_equations(*post_discretisation*)

Check that the algebraic equations are well-posed. After discretisation, there must be at least one StateVector in each algebraic equation.

check_discretised_or_discretise_inplace_if_0D()

Discretise model if it isn't already discretised This only works with purely 0D models, as otherwise the mesh and spatial method should be specified by the user

check_ics_bcs()

Check that the initial and boundary conditions are well-posed.

check_no_repeated_keys()

Check that no equation keys are repeated.

check_well_determined(*post_discretisation*)

Check that the model is not under- or over-determined.

check_well_posedness(*post_discretisation=False*)

Check that the model is well-posed by executing the following tests: - Model is not over- or underdetermined, by comparing keys and equations in rhs and algebraic. Overdetermined if more equations than variables, underdetermined if more variables than equations. - There is an initial condition in self.initial_conditions for each variable/equation pair in self.rhs - There are appropriate boundary conditions in self.boundary_conditions for each variable/equation pair in self.rhs and self.algebraic

Parameters

post_discretisation
[boolean] A flag indicating tests to be skipped after discretisation

property default_geometry

Override: corrects the geometry for half-cells.

property default_solver

Return default solver based on whether model is ODE/DAE or algebraic

classmethod deserialise(*properties: dict*)

Create a model instance from a serialised object.

export_casadi_objects(*variable_names*, *input_parameter_order=None*)

Export the constituent parts of the model (rhs, algebraic, initial conditions, etc) as casadi objects.

Parameters

variable_names
[list] Variables to be exported alongside the model structure

input_parameter_order
[list, optional] Order in which the input parameters should be stacked. If input_parameter_order=None and len(self.input_parameters) > 1, a ValueError is raised (this helps to avoid accidentally using the wrong order)

Returns

casadi_dict
[dict] Dictionary of {str: casadi object} pairs representing the model in casadi format

generate(*filename*, *variable_names*, *input_parameter_order=None*, *cg_options=None*)

Generate the model in C, using CasADi.

Parameters

filename
[str] Name of the file to which to save the code

variable_names
[list] Variables to be exported alongside the model structure

input_parameter_order
[list, optional] Order in which the input parameters should be stacked. If input_parameter_order=None and len(self.input_parameters) > 1, a ValueError is raised (this helps to avoid accidentally using the wrong order)

cg_options
[dict] Options to pass to the code generator. See https://web.casadi.org/docs/#generating-c-code

get_parameter_info(*by_submodel=False*)

Extracts the parameter information and returns it as a dictionary. To get a list of all parameter-like objects without extra information, use `model.parameters`.

Parameters

by_submodel
[bool, optional] Whether to return the parameter info sub-model wise or not (default False)

info(*symbol_name*)

Provides helpful summary information for a symbol.

Parameters

parameter_name : str

property input_parameters

Returns all the input parameters in the model

latexify(*filename=None*, *newline=True*, *output_variables=None*)

Converts all model equations in latex.

Parameters

filename: str (optional)
Accepted file formats - any image format, pdf and tex Default is None, When None returns all model equations in latex If not None, returns all model equations in given file format.

newline: bool (optional)
Default is True, If True, returns every equation in a new line. If False, returns the list of all the equations.

Load model >>> model = pybamm.lithium_ion.SPM()

This will returns all model equations in png >>> model.latexify("equations.png") # doctest: +SKIP

This will return all the model equations in latex >>> model.latexify() # doctest: +SKIP

This will return the list of all the model equations >>> model.latexify(newline=False) # doctest: +SKIP

This will return first five model equations >>> model.latexify(newline=False)[1:5] # doctest: +SKIP

property length_scales

Non-dimensionalization length scales.

new_copy(*build=True*)

Create an empty copy with identical options.

Parameters

build – If True, the new model gets built right away. This is the default behavior. If set to False, it remains as symbols.

Returns

The copy of this model.

property param

Contains all the relevant parameters for this model.

property parameters

Returns all the parameters in the model

print_parameter_info(*by_submodel=False*)

Print parameter information in a formatted table from a dictionary of parameters

Parameters

by_submodel

[bool, optional] Whether to print the parameter info sub-model wise or not (default False)

process_parameters_and_discretise(*symbol*, *parameter_values*, *disc*)

Process parameters and discretise a symbol using supplied parameter values and discretisation. Note: care should be taken if using spatial operators on dimensional symbols. Operators in pybamm are written in non-dimensional form, so may need to be scaled by the appropriate length scale. It is recommended to use this method on non-dimensional symbols.

Parameters

symbol

[`pybamm.Symbol`] Symbol to be processed

parameter_values

[`pybamm.ParameterValues`] The parameter values to use during processing

disc

[`pybamm.Discretisation`] The discrisation to use

Returns

pybamm.Symbol
Processed symbol

pybamm_control

Current is fixed if False and a variable if True.

save_model(*filename=None*, *mesh=None*, *variables=None*)

Write out a discretised model to a JSON file

Parameters

filename: str, optional The desired name of the JSON file. If no name is provided, one will be created based on the model name, and the current datetime.

set_degradation_variables()

Set variables that quantify degradation. This function is overriden by the base battery models

set_external_circuit_submodel()

Define how the external circuit defines the boundary conditions for the model, e.g. (not necessarily constant-) current, voltage, etc

set_initial_conditions_from(*solution*, *inplace=True*, *return_type='model'*)

Update initial conditions with the final states from a Solution object or from a dictionary. This assumes that, for each variable in self.initial_conditions, there is a corresponding variable in the solution with the same name and size.

Parameters

solution
[pybamm.Solution, or dict] The solution to use to initialize the model

inplace
[bool, optional] Whether to modify the model inplace or create a new model (default True)

return_type
[str, optional] Whether to return the model (default) or initial conditions ("ics")

set_soc_variables()

Set variables relating to the state of charge. This function is overriden by the base battery models

set_standard_output_variables()

Adds "the horizontal axis" to self.variables.

Don't use the super() version of this function, as it introduces keys with integer values in self.variables.

set_voltage_variables()

Adds voltage-specific variables to self.variables.

Override this inherited function, since it adds superfluous variables.

property timescale

Non-dimensionalization timescale.

update(**submodels*)

Update model to add new physics from submodels

Parameters

submodel
[iterable of `pybamm.BaseModel`] The submodels from which to create new model

property variables_and_events

Returns variables and events in a single dictionary

ep_bolfi.models.electrolyte.Electrolyte_internal

class ep_bolfi.models.electrolyte.**Electrolyte_internal**(*param*, *pybamm_control=False*, *options={}*, *build=True*)

Bases: `BaseSubModel`

Defining equations for a symmetric Li cell with electrolyte.

Reference

SG Marquis, V Sulzer, R Timms, CP Please and SJ Chapman. "An asymptotic derivation of a single particle model with electrolyte". Journal of The Electrochemical Society, 166(15):A3693–A3706, 2019

__init__(*param*, *pybamm_control=False*, *options={}*, *build=True*)

Sets the model properties.

Parameters

- **param** – A class containing all the relevant parameters for this model. For example, models.standard_parameters represents a valid choice for this parameter.
- **pybamm_control** – Per default False, which indicates that the current is given as a function. If set to True, this model is compatible with PyBaMM experiments, e.g. CC-CV simulations. The current is then a variable and it or voltage can be fixed functions.
- **options** – Not used; only here for compatibility with the base class.
- **build** – Not used; only here for compatibility with the base class.

Methods

__init__(param[, pybamm_control, options, build])	Sets the model properties.
build_coupled_variables()	
build_fundamental()	
build_model()	
build_model_equations()	
check_algebraic_equations(post_discretisation)	Check that the algebraic equations are well-posed.
check_and_combine_dict(dict1, dict2)	

continues on next page

Table 5 – continued from previous page

check_discretised_or_discretise_in-place_if_0D()	Discretise model if it isn't already discretised This only works with purely 0D models, as otherwise the mesh and spatial method should be specified by the user
check_for_time_derivatives()	
check_ics_bcs()	Check that the initial and boundary conditions are well-posed.
check_no_repeated_keys()	Check that no equation keys are repeated.
check_variables()	
check_well_determined(post_discretisation)	Check that the model is not under- or over-determined.
check_well_posedness([post_discretisation])	Check that the model is well-posed by executing the following tests: - Model is not over- or underdetermined, by comparing keys and equations in rhs and algebraic.
deserialise(properties)	Create a model instance from a serialised object.
export_casadi_objects(variable_names[, ...])	Export the constituent parts of the model (rhs, algebraic, initial conditions, etc) as casadi objects.
generate(filename, variable_names[, ...])	Generate the model in C, using CasADi.
generic_deserialise(instance, properties)	
get_coupled_variables(variables)	Builds all model symbols that rely on other models.
get_fundamental_variables()	Builds all relevant model variables' symbols.
get_parameter_info([by_submodel])	Extracts the parameter information and returns it as a dictionary.
info(symbol_name)	Provides helpful summary information for a symbol.
latexify([filename, newline, output_variables])	Converts all model equations in latex.
new_copy()	Creates a copy of the model, explicitly copying all the mutable attributes to avoid issues with shared objects.
print_parameter_info([by_submodel])	Print parameter information in a formatted table from a dictionary of parameters
process_parameters_and_discre-tise(symbol, ...)	Process parameters and discretise a symbol using supplied parameter values and discretisation.
save_model([filename, mesh, variables])	Write out a discretised model to a JSON file
set_algebraic(variables)	Sets up the algebraic equations in self.algebraic.
set_boundary_conditions(variables)	Sets the (self.)boundary(_)conditions.
set_events(variables)	Sets up the termination switches in self.events.
set_initial_conditions(variables)	Sets the (self.)initial(_)conditions.
set_initial_conditions_from(solution[, ...])	Update initial conditions with the final states from a Solution object or from a dictionary.
set_phase(phase)	
set_rhs(variables)	Sets up the right-hand-side equations in self.rhs.
update(*submodels)	Update model to add new physics from submodels
variable_names()	

Attributes

algebraic
boundary_conditions
concatenated_algebraic
concatenated_initial_conditions
concatenated_rhs
default_geometry
default_parameter_values

continues on next page

Table 6 – continued from previous page

default_quick_plot_variables	
default_solver	Return default solver based on whether model is ODE/DAE or algebraic
default_spatial_methods	
default_submesh_types	
default_var_pts	
domain	
domain_Domain	
events	
geometry	
initial_conditions	
input_parameters	Returns all the input parameters in the model
jacobian	
jacobian_algebraic	
jacobian_rhs	
length_scales	
mass_matrix	
mass_matrix_inv	
name	
options	
param	
parameters	Returns all the parameters in the model
rhs	
timescale	
variables	
variables_and_events	Returns variables and events in a single dictionary
pybamm_control	Current is fixed if False and a variable if True.

check_algebraic_equations(*post_discretisation*)

Check that the algebraic equations are well-posed. After discretisation, there must be at least one StateVector in each algebraic equation.

check_discretised_or_discretise_inplace_if_0D()

Discretise model if it isn't already discretised This only works with purely 0D models, as otherwise the mesh and spatial method should be specified by the user

check_ics_bcs()

Check that the initial and boundary conditions are well-posed.

check_no_repeated_keys()

Check that no equation keys are repeated.

check_well_determined(*post_discretisation*)

Check that the model is not under- or over-determined.

check_well_posedness(*post_discretisation=False*)

Check that the model is well-posed by executing the following tests: - Model is not over- or underdetermined, by comparing keys and equations in rhs and algebraic. Overdetermined if more equations than variables, underdetermined if more variables than equations. - There is an initial condition in self.initial_conditions for each variable/equation pair in self.rhs - There are appropriate boundary conditions in self.boundary_conditions for each variable/equation pair in self.rhs and self.algebraic

Parameters

post_discretisation
[boolean] A flag indicating tests to be skipped after discretisation

property default_solver

Return default solver based on whether model is ODE/DAE or algebraic

classmethod deserialise(*properties: dict*)

Create a model instance from a serialised object.

export_casadi_objects(*variable_names*, *input_parameter_order=None*)

Export the constituent parts of the model (rhs, algebraic, initial conditions, etc) as casadi objects.

Parameters

variable_names
[list] Variables to be exported alongside the model structure

input_parameter_order
[list, optional] Order in which the input parameters should be stacked. If input_parameter_order=None and len(self.input_parameters) > 1, a ValueError is raised (this helps to avoid accidentally using the wrong order)

Returns

casadi_dict
[dict] Dictionary of {str: casadi object} pairs representing the model in casadi format

generate(*filename*, *variable_names*, *input_parameter_order=None*, *cg_options=None*)

Generate the model in C, using CasADi.

Parameters

filename
[str] Name of the file to which to save the code

variable_names
[list] Variables to be exported alongside the model structure

input_parameter_order
[list, optional] Order in which the input parameters should be stacked. If input_parameter_order=None and len(self.input_parameters) > 1, a ValueError is raised (this helps to avoid accidentally using the wrong order)

cg_options
[dict] Options to pass to the code generator. See https://web.casadi.org/docs/#generating-c-code

get_coupled_variables(*variables*)

Builds all model symbols that rely on other models.

Parameters
variables – A dictionary containing at least all variable symbols that are required for the variable symbols built here.

Returns
A dictionary with the new variables' names as keys and their symbols (of type pybamm.Symbol) as values.

get_fundamental_variables()

Builds all relevant model variables' symbols.

Returns
A dictionary with the variables' names as keys and their symbols (of type pybamm.Symbol) as values.

get_parameter_info(*by_submodel=False*)

Extracts the parameter information and returns it as a dictionary. To get a list of all parameter-like objects without extra information, use `model.parameters`.

info(*symbol_name*)

Provides helpful summary information for a symbol.

Parameters

parameter_name : str

property input_parameters

Returns all the input parameters in the model

latexify(*filename=None*, *newline=True*, *output_variables=None*)

Converts all model equations in latex.

Parameters

filename: str (optional)
Accepted file formats - any image format, pdf and tex Default is None, When None returns all model equations in latex If not None, returns all model equations in given file format.

newline: bool (optional)
Default is True, If True, returns every equation in a new line. If False, returns the list of all the equations.

Load model >>> model = pybamm.lithium_ion.SPM()

This will returns all model equations in png >>> model.latexify("equations.png") # doctest: +SKIP

This will return all the model equations in latex >>> model.latexify() # doctest: +SKIP

This will return the list of all the model equations >>> model.latexify(newline=False) # doctest: +SKIP

This will return first five model equations >>> model.latexify(newline=False)[1:5] # doctest: +SKIP

new_copy()

Creates a copy of the model, explicitly copying all the mutable attributes to avoid issues with shared objects.

property parameters

Returns all the parameters in the model

print_parameter_info(*by_submodel=False*)

Print parameter information in a formatted table from a dictionary of parameters

Parameters

by_submodel
[bool, optional] Whether to print the parameter info sub-model wise or not (default False)

process_parameters_and_discretise(*symbol*, *parameter_values*, *disc*)

Process parameters and discretise a symbol using supplied parameter values and discretisation. Note: care should be taken if using spatial operators on dimensional symbols. Operators in pybamm are written in non-dimensional form, so may need to be scaled by the appropriate length scale. It is recommended to use this method on non-dimensional symbols.

Parameters

symbol
[`pybamm.Symbol`] Symbol to be processed

parameter_values
[`pybamm.ParameterValues`] The parameter values to use during processing

disc
[`pybamm.Discretisation`] The discrisation to use

Returns

pybamm.Symbol
Processed symbol

pybamm_control

Current is fixed if False and a variable if True.

save_model(*filename=None*, *mesh=None*, *variables=None*)

Write out a discretised model to a JSON file

Parameters

filename: str, optional The desired name of the JSON file. If no name is provided, one will be created based on the model name, and the current datetime.

set_algebraic(*variables*)

Sets up the algebraic equations in self.algebraic.

set_boundary_conditions(*variables*)

Sets the (self.)boundary(_)conditions.

Parameters
variables – A dictionary containing at least all variable symbols that are required for the variable symbols built here.

Returns
A dictionary with the solvable variables' names as keys and their boundary conditions as values.

set_events(*variables*)

Sets up the termination switches in self.events.

Parameters
variables – A dictionary containing at least all variable symbols that are required for the variable symbols built here.

Returns
A dictionary with the event names as keys and their trigger conditions as values.

set_initial_conditions(*variables*)

Sets the (self.)initial(_)conditions.

Parameters
variables – A dictionary containing at least all variable symbols that are required for the variable symbols built here.

Returns
A dictionary with the solvable variables' names as keys and their initial conditions as values.

set_initial_conditions_from(*solution*, *inplace=True*, *return_type='model'*)

Update initial conditions with the final states from a Solution object or from a dictionary. This assumes that, for each variable in self.initial_conditions, there is a corresponding variable in the solution with the same name and size.

Parameters

solution
[pybamm.Solution, or dict] The solution to use to initialize the model

inplace
[bool, optional] Whether to modify the model inplace or create a new model (default True)

return_type
[str, optional] Whether to return the model (default) or initial conditions ("ics")

set_rhs(*variables*)

Sets up the right-hand-side equations in self.rhs.

Parameters
variables – A dictionary containing at least all variable symbols that are required for the variable symbols built here.

Returns
A dictionary with the solvable variables' names as keys and their right-hand-sides as values.

update(**submodels*)

Update model to add new physics from submodels

Parameters

submodel
[iterable of `pybamm.BaseModel`] The submodels from which to create new model

property variables_and_events
Returns variables and events in a single dictionary

ep_bolfi.models.equivalent_circuits

Functions

condense(eq, *x)	collapse additive/multiplicative constants into single variables, returning condensed expression and replacement values.
parallel(*ecms)	
series(*ecms)	

ep_bolfi.models.equivalent_circuits.condense

ep_bolfi.models.equivalent_circuits.**condense**(*eq*, **x*)

collapse additive/multiplicative constants into single variables, returning condensed expression and replacement values.

https://stackoverflow.com/questions/71315789/optimize-sympy-expression- evaluation-by-combining-as-many-free-symbols-as-poss
by smichr under CC-BY-SA 4.0: https://creativecommons.org/licenses/by-sa/4.0/

Examples

Simple constants are left unchanged

```
>>> condense(2*x + 2, x)
(2*x + 2, {})
```

More complex constants are replaced by a single variable

```
>>> first = condense(eq, x); first
(c6*(c5 - 2*sqrt(d*(c4 + x))), {c4: a*b - c - e, c6: 1/(b - 1),
 c5: a*b*c**2})
```

If a condensed expression is expanded, there may be more simplification possible:

```
>>> second = condense(first[0].expand(), x); second
(c0 + c2*sqrt(c1 + d*x), {c1: c4*d, c2: -2*c6, c0: c5*c6})
>>> full_reps = {k: v.xreplace(first[1]) for k, v in second[1].items()};
    full_reps
{c1: d*(a*b - c - e), c2: -2/(b - 1), c0: a*b*c**2/(b - 1)}
```

More than 1 variable can be designated:

```
>>> condense(eq, c, e)
(c4*(c**2*c1 - 2*sqrt(d*(-c + c2 - e))), {c4: 1/(b - 1), c1: a*b,
 c2: a*b + x})
```

ep_bolfi.models.equivalent_circuits.parallel

ep_bolfi.models.equivalent_circuits.**parallel**(**ecms*)

ep_bolfi.models.equivalent_circuits.series

ep_bolfi.models.equivalent_circuits.**series**(**ecms*)

Classes

C(capacitance)	
ECM([EC, ECM_parameters, Q_global_index, ...])	
L(inductance)	
Q(cpe, exponent)	Constant Phase Element (CPE).
R(resistance)	
RC_chain(ECM_parameters[, chain_length])	RC pairs in series with a resistor.
SCR(warburg)	SCR circuit for cathodes.
SCRF(warburg)	SCRF circuit for anodes with SEI.
Two_RC_Optimized_for_Torch(omega[, ...])	
W(coefficient)	Warburg with semi-infinite diffusion condition.
aluminium_electrode()	Models an aluminium electrode incl.
aluminium_electrode_variant()	Can fit the same spectrum, but with less meaningful parameters.
debye(ECM_parameters)	Debye circuit for ideally blocking electrodes.
debye_variant(ECM_parameters)	Often used variant of the Debye circuit.
randles(warburg)	Randles circuit.
randles_variant(warburg)	Randles circuit variant.
warburg_open(coefficient, characteristic)	Also known as finite-space Warburg.
warburg_short(coefficient, characteristic)	Also known as finite-length Warburg.
wrong_randles(warburg)	Wrong Randles circuit.
wrong_randles_variant(warburg, exponent)	Variant of the wrong Randles circuit.

ep_bolfi.models.equivalent_circuits.C

class ep_bolfi.models.equivalent_circuits.**C**(*capacitance*)

Bases: *ECM*

__init__(*capacitance*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(capacitance)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.ECM

class ep_bolfi.models.equivalent_circuits.**ECM**(*EC=None*, *ECM_parameters=None*, *Q_global_index=None*, *I_global_index=None*, *U_global_index=None*, *ECM=None*)

Bases: object

__init__(*EC=None*, *ECM_parameters=None*, *Q_global_index=None*, *I_global_index=None*, *U_global_index=None*, *ECM=None*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__([EC, ECM_parameters, ...])	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.L

class ep_bolfi.models.equivalent_circuits.**L**(*inductance*)

Bases: *ECM*

__init__(*inductance*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(inductance)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.Q

class ep_bolfi.models.equivalent_circuits.**Q**(*cpe*, *exponent*)

Bases: *ECM*

Constant Phase Element (CPE). exponent is in (0, 1].

Impedance is phase-shifted by a constant factor n. Nyquist plot looks like C, but rotated by (1 - n) * 90°.

Models imperfect capacitors. Hence, in general use values >> 0.5. Origins:

- surface roughness (fractal dimension),
- distribution of reaction rates (varying activation energies),
- varying thickness or composition of (surface) coatings,
- non-uniform current distribution (n decreases towards electrode edge).

In RQ elements, the center of the semicircle rotates below the real axis. A “true” capacitance C may be obtained via $RQ = \tau^n = (RC)^n$.

__init__(*cpe*, *exponent*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(cpe, exponent)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.R

class ep_bolfi.models.equivalent_circuits.**R**(*resistance*)

Bases: *ECM*

__init__(*resistance*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(resistance)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.RC_chain

class ep_bolfi.models.equivalent_circuits.**RC_chain**(*ECM_parameters*, *chain_length=1*)

Bases: *ECM*

RC pairs in series with a resistor.

Keyword arguments for *_rhs: R_0, R_1, C_1, R_2, C_2, …

__init__(*ECM_parameters*, *chain_length=1*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(ECM_parameters[, chain_length])	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

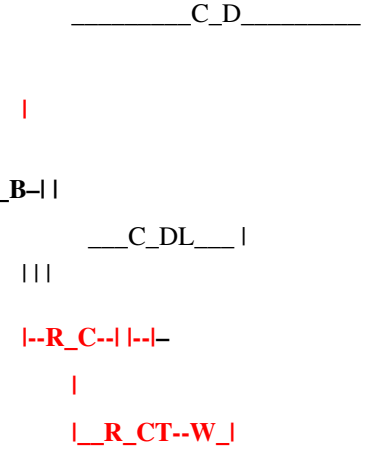
ep_bolfi.models.equivalent_circuits.SCR

class ep_bolfi.models.equivalent_circuits.**SCR**(*warburg*)

Bases: *ECM*

SCR circuit for cathodes. Stems from a simplification of a TLM.

R_B: bulk electrolyte C_D: dielectric capacitance R_C: resistance between current collector and electrode C_DL: double-layer R_CT: charge-transfer resistance W: Warburg for mass-transport



__init__(*warburg*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(warburg)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

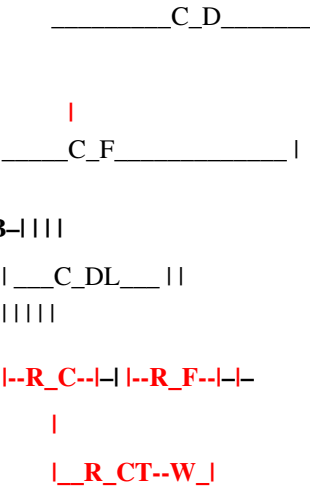
ep_bolfi.models.equivalent_circuits.SCRF

class ep_bolfi.models.equivalent_circuits.**SCRF**(*warburg*)

Bases: *ECM*

SCRF circuit for anodes with SEI. Stems from a simplification of a TLM.

R_B: bulk electrolyte C_D: dielectric capacitance R_C: resistance between current collector and electrode C_F: capacitance of the passivating film (i.e. SEI) C_DL: double-layer R_CT: charge-transfer resistance W: Warburg for mass-transport R_F: resistance of the passivating film (i.e. SEI)



__init__(*warburg*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(warburg)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.Two_RC_Optimized_for_Torch

class ep_bolfi.models.equivalent_circuits.**Two_RC_Optimized_for_Torch**(*omega*, *true_parameters=None*, *measured_real_part=None*, *measured_imaginary_part=None*)

Bases: object

__init__(*omega*, *true_parameters=None*, *measured_real_part=None*, *measured_imaginary_part=None*)

@param omega
A torch.Tensor of the angular frequencies in rad / s.

@param true_parameters
If given, synthetic data will be generated for measured Z.

@param measured_real_part
If 'true_parameters' is not set, you may give experimental data directly instead.

@param measured_imaginary_part
If 'true_parameters' is not set, you may give experimental data directly instead.

Methods

__init__(omega[, true_parameters, ...])	@param omega
circuit_elements(theta)	
imaginary_part(Rt, r1, t1, r2, t2)	Returns:
normalised_input(tau)	Args:
real_part(Rt, r0, r1, t1, r2, t2)	Returns:
unnormalise_tau(tau)	@param tau: torch.tensor, time constant tau in log space;

imaginary_part(*Rt*, *r1*, *t1*, *r2*, *t2*)

Returns:

- Z.imaginary: torch.tensor, imaginary part of impedance spectrum

normalised_input(*tau*)

Args:

- **tau: torch.tensor, time constant tau in raw space;**
  tau = omega * t_i

Returns:

- **tau: torch.tensor, time constant tau in log space;**
  tau = ln(omega * t_i)

real_part(*Rt*, *r0*, *r1*, *t1*, *r2*, *t2*)

Returns:

- Z.real: torch.tensor, real part of impedance spectrum

unnormalise_tau(*tau*)

@param tau: torch.tensor, time constant tau in log space;
tau = ln(omega * t_i)

Returns:

- **tau: torch.tensor, time constant tau in raw space;**
  tau = omega * t_i

ep_bolfi.models.equivalent_circuits.W

class ep_bolfi.models.equivalent_circuits.**W**(*coefficient*)

Bases: *ECM*

Warburg with semi-infinite diffusion condition.

Represents an infinite transmission line of RC elements. “coefficient” refers to the Warburg coefficient σ. “characteristic” does nothing, as there is no characteristic time-scale.

σ = RT / (n^2 F^2 A sqrt(2)) * 1 / (
sqrt(D_O) c_O^(b) sqrt(D_R) c_R^(b)

) with D_x being the diffusion coefficients of reduced and oxidized species and c_x bulk concentrations (^(b)) of reduced and oxidized species.

__init__(*coefficient*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(coefficient)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.aluminium_electrode

class ep_bolfi.models.equivalent_circuits.**aluminium_electrode**

Bases: *ECM*

Models an aluminium electrode incl. adsorption process.

R_s: resistance of the electrolyte (measurements are typically in frequencies too low to see its capacitance)

C_1: double-layer capacitance. R_1: unclear. L_1: models the loop attributed to the adsorption process. R_2: resistance of the adsorption process. C: charge-transfer capacitance. R_3: charge-transfer resistance.

__init__()

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__()	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.aluminium_electrode_variant

class ep_bolfi.models.equivalent_circuits.**aluminium_electrode_variant**

Bases: *ECM*

Can fit the same spectrum, but with less meaningful parameters.

R_s: resistance of the electrolyte (measurements are typically
in frequencies too low to see its capacitance)

C_1: double-layer capacitance. R_1: unclear. L_1: models the loop attributed to the adsorption process. R_2: resistance of the adsorption process. C: charge-transfer capacitance. R_3: charge-transfer resistance.

__init__()

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__()	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.debye

class ep_bolfi.models.equivalent_circuits.**debye**(*ECM_parameters*)

Bases: *ECM*

Debye circuit for ideally blocking electrodes.

B: bulk electrolyte D: dielectric capacitance DL: double-layer

__R_B__

|

–| |–C_DL–

|

|__C_D__|

__init__(*ECM_parameters*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(ECM_parameters)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.debye_variant

class ep_bolfi.models.equivalent_circuits.**debye_variant**(*ECM_parameters*)

Bases: *ECM*

Often used variant of the Debye circuit. Valid as well.

__R_B__C_DL__

|

–| |–

|

|_____C_D_____|

__init__(*ECM_parameters*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(ECM_parameters)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.randles

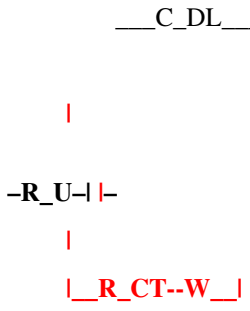
class ep_bolfi.models.equivalent_circuits.**randles**(*warburg*)

Bases: *ECM*

Randles circuit. Very similar to SPM impedance with open Warburg.

Pass one of the Warburg classes to choose the right Warburg.

U: uncompensated resistance between reference electrode and metal surface DL: double-layer CT: charge-transfer resistance W: Warburg for mass-transport



__init__(*warburg*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(warburg)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.randles_variant

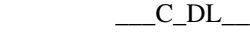
class ep_bolfi.models.equivalent_circuits.**randles_variant**(*warburg*)

Bases: *ECM*

Randles circuit variant.

Pass one of the Warburg classes to choose the right Warburg.

U: uncompensated resistance between reference electrode and metal surface DL: double-layer CT: charge-transfer resistance W: Warburg for mass-transport



–R_U–| |–C_X–

|__R_CT--W__|

__init__(*warburg*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(warburg)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.warburg_open

class ep_bolfi.models.equivalent_circuits.**warburg_open**(*coefficient*, *characteristic*)

Bases: *ECM*

Also known as finite-space Warburg. Closes with a capacitor.

One plate of the closing capacitor may be thought of as the counter electrode. Nyquist plot diverges to imaginary infinity.

“coefficient” refers to the Warburg coefficient. “characteristic” refers to the ratio between thickness of the diffusion layer and the square-root of the diffusivity.

__init__(*coefficient*, *characteristic*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(coefficient, characteristic)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.warburg_short

class ep_bolfi.models.equivalent_circuits.**warburg_short**(*coefficient*, *characteristic*)

Bases: *ECM*

Also known as finite-length Warburg. Closes with a resistor.

Represents diffusion layers with controlled thickness. Nyquist plot goes from the 45° line into a quarter circle back to the real axis.

“coefficient” refers to the Warburg coefficient. “characteristic” refers to the ratio between thickness of the diffusion layer and the square-root of the diffusivity.

__init__(*coefficient*, *characteristic*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(coefficient, characteristic)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.wrong_randles

class ep_bolfi.models.equivalent_circuits.**wrong_randles**(*warburg*)

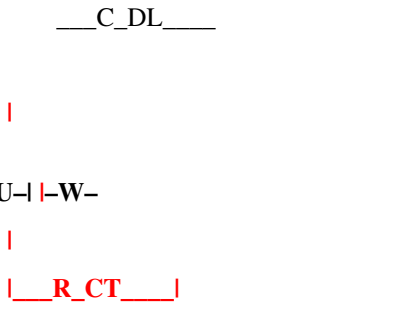
Bases: *ECM*

Wrong Randles circuit. Often used even if it is not applicable.

Would imply that CT resistance and mass transport are not coupled (wrong), while CT resistance and double-layer capacitance are (also wrong).

Pass one of the Warburg classes to choose the right Warburg.

U: uncompensated resistance between reference electrode and metal surface DL: double-layer CT: charge-transfer resistance W: Warburg for mass-transport



__init__(*warburg*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(warburg)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.equivalent_circuits.wrong_randles_variant

class ep_bolfi.models.equivalent_circuits.**wrong_randles_variant**(*warburg*, *exponent*)

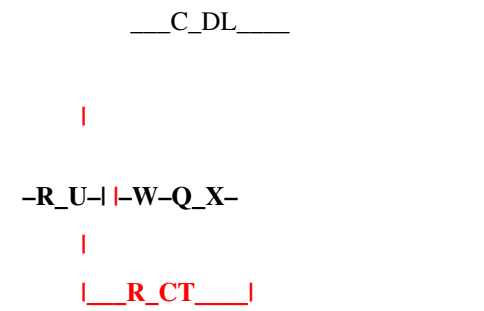
Bases: *ECM*

Variant of the wrong Randles circuit.

Contains a redundant CPE.

Pass one of the Warburg classes to choose the right Warburg.

U: uncompensated resistance between reference electrode and metal surface DL: double-layer CT: charge-transfer resistance W: Warburg for mass-transport



__init__(*warburg*, *exponent*)

Methods

I_rhs(t, state, U_derivatives)	
U_rhs(t, state, I_derivatives)	
__init__(warburg, exponent)	
calculate_I_ode_state_for_new_control(...)	
calculate_U_ode_state_for_new_control(...)	
check_parameters_validity(reference_list)	
lambdify_ECM()	
update_I_ode_state_to_new_input(old_I_state, ...)	
update_U_ode_state_to_new_input(old_U_state, ...)	

ep_bolfi.models.solversetup

This file eases the setup and simulation of PyBaMM battery models.

Functions

auto_var_pts(x_n, x_s, x_p, r_n, r_p[, y, ...])	Utility function for setting the discretization density.
simulation_setup(model, operation_input, ...)	Processes the model and returns a runnable solver.
solver_setup(model, parameters, ...[, ...])	Processes the model and returns a runnable solver.
spectral_mesh_pts_and_method(order_s_n, ...)	Utility function for default mesh and spatial methods.

ep_bolfi.models.solversetup.auto_var_pts

ep_bolfi.models.solversetup.**auto_var_pts**(*x_n*, *x_s*, *x_p*, *r_n*, *r_p*, *y=1*, *z=1*, *R_n=1*, *R_p=1*)

Utility function for setting the discretization density.

Parameters

- **x_n** – The number of voxels for the electrolyte in the anode.
- **x_s** – The number of voxels for the electrolyte in the separator.
- **x_p** – The number of voxels for the electrolyte in the cathode.
- **r_n** – The number of voxels for each anode representative particle.
- **r_p** – The number of voxels for each cathode representative particle.
- **y** – Used by PyBaMM for spatially resolved current collectors. Don't change the default (1) unless the model supports it.
- **z** – Used by PyBaMM for spatially resolved current collectors. Don't change the default (1) unless the model supports it.
- **R_n** – Used by PyBaMM for particle size distributions. Don't change the default (1) unless the model supports it.
- **R_p** – Used by PyBaMM for particle size distributions. Don't change the default (1) unless the model supports it.

Returns

A discretization dictionary that can be used with PyBaMM models.

ep_bolfi.models.solversetup.simulation_setup

ep_bolfi.models.solversetup.**simulation_setup**(*model*, *operation_input*, *parameters*, *submesh_types*, *var_pts*, *spatial_methods*, *geometry=None*, *reltol=1e-06*, *abstol=1e-06*, *root_tol=0.001*, *dt_max=None*, *free_parameters=[]*, *verbose=False*, *logging_file=None*)

Processes the model and returns a runnable solver.

In contrast to `solversetup.solver_setup`, this allows for a more verbose description of the operating conditions and should be preferred.

Parameters

- **model** – A PyBaMM model. Use one of the models in this folder.

- **operation_input** – A list of strings which describe the operating conditions. These exactly match the PyBaMM usage for pybamm.Experiment. Examples: “Hold at 4 V for 60 s”, “Discharge at 1 A for 30 s”, “Rest for 1800 s”, “Charge at 1 C for 30 s”.
- **parameters** – The parameters that the model requires as a dictionary. Please refer to models.standard_parameters for the names or adapt one of the examples in parameters.models.
- **submesh_types** – A dictionary of the meshes to be used. The keys have to match the geometry names in the model. Use #spectral_mesh_and_method as reference or a shortcut.
- **var_pts** – A dictionary giving the number of discretization volumes. Since the keys have to be special variables determined by PyBaMM, use #auto_var_pts as a shortcut.
- **spatial_methods** – A dictionary of the spatial methods to be used. The keys have to match the geometry names in the model. Use #spectral_mesh_and_method as reference or a shortcut.
- **geometry** – The geometry of the model in dictionary form. Usually, model.default_geometry is sufficient, which is the default.
- **reltol** – The relative tolerance that the Casadi solver should use. Default is 1e-6.
- **abstol** – The absolute tolerance that the Casadi solver should use. Default is 1e-6.
- **root_tol** – The tolerance for rootfinding that the Casadi solver should use. Default is 1e-3.
- **dt_max** – The maximum timestep size for the Casadi solver in seconds. Default is chosen by PyBaMM.
- **free_parameters** – A list of parameter names that shall be input later. They may be given to the solve() function of the returned Simulation object as a dictionary to the keyword parameter “inputs” with the names as keys and the values of the parameters as values. DO NOT USE GEOMETRICAL PARAMETERS, THEY WILL CRASH THE MESH. Instead, just use this function with a complete set of parameters where the relevant parameters are changed.
- **verbose** – The default (False) sets the PyBaMM flag to only show warnings. True will show the details of preprocessing and the runtime of the solver. This applies globally, so don’t set this to True if running simulations in parallel.
- **logging_file** – Optional name of a file to store the logs in.

Returns

A 2-tuple of a pybamm.Simulation.solve call that runs the simulation when called, and the proper callback for the logging file if specified (else None). Please use solve(check_model=False) if it doesn’t work properly with redundant model checks in place.

ep_bolfi.models.solversetup.solver_setup

ep_bolfi.models.solversetup.**solver_setup**(*model*, *parameters*, *submesh_types*, *var_pts*, *spatial_methods*, *geometry=None*, *reltol=1e-06*, *abstol=1e-06*, *root_tol=0.001*, *dt_max=None*, *free_parameters=[]*, *verbose=False*, *logging_file=None*)

Processes the model and returns a runnable solver.

Parameters

- **model** – A PyBaMM model. Use one of the models in this folder.
- **parameters** – The parameters that the model requires as a dictionary. Please refer to models.standard_parameters for the names or adapt one of the examples in parameters.models.

- **submesh_types** – A dictionary of the meshes to be used. The keys have to match the geometry names in the model. Use #spectral_mesh_and_method as reference or a shortcut.
- **var_pts** – A dictionary giving the number of discretization volumes. Since the keys have to be special variables determined by PyBaMM, use #auto_var_pts as a shortcut.
- **spatial_methods** – A dictionary of the spatial methods to be used. The keys have to match the geometry names in the model. Use #spectral_mesh_and_method as reference or a shortcut.
- **geometry** – The geometry of the model in dictionary form. Usually, model.default_geometry is sufficient, which is the default.
- **reltol** – The relative tolerance that the Casadi solver shall use. Default is 1e-6.
- **abstol** – The absolute tolerance that the Casadi solver shall use. Default is 1e-6.
- **root_tol** – The tolerance for rootfinding that the Casadi solver shall use. Default is 1e-3.
- **dt_max** – The maximum timestep size for the Casadi solver in seconds. Default is chosen by PyBaMM.
- **free_parameters** – A list of parameter names that shall be input later. They may be given to the returned lambda function as a dictionary with the names as keys and the values of the parameters as values. DO NOT USE GEOMETRICAL PARAMETERS, THEY WILL CRASH THE MESH. Instead, just use this function with a complete set of parameters where the relevant parameters are changed.
- **verbose** – The default (False) sets the PyBaMM flag to only show warnings. True will show the details of preprocessing and the runtime of the solver. This applies globally, so don't set this to True if running simulations in parallel.
- **logging_file** – Optional name of a file to store the logs in.

Returns

A lambda function that takes a numpy.array of timepoints to evaluate and runs the Casadi solver for those. Optionally takes a dictionary of parameters as specified by "free_parameters".

ep_bolfi.models.solversetup.spectral_mesh_pts_and_method

ep_bolfi.models.solversetup.**spectral_mesh_pts_and_method**(*order_s_n*, *order_s_p*, *order_e*, *volumes_e_n=1*, *volumes_e_s=1*, *volumes_e_p=1*, *halfcell=False*)

Utility function for default mesh and spatial methods.

Only returns Spectral Volume mesh and spatial methods.

Parameters

- **order_s_n** – The order of the anode particles Spectral Volumes.
- **order_s_p** – The order of the anode particles Spectral Volumes.
- **order_e** – The order of the anode, separator and cathode electrolyte Spectral Volumes. These have to be the same, since the corresponding meshes get concatenated.
- **volumes_e_n** – The # of Spectral Volumes to use for the anode electrolyte. This is useful to have different resolutions for each part.
- **volumes_e_s** – The # of Spectral Volumes to use for the separator electrolyte.

- **volumes_e_p** – The # of Spectral Volumes to use for the cathode electrolyte.
- **halfcell** – Default is False, which sets up the mesh and spatial methods for a full-cell setup. Set it to True for a half-cell setup.

Returns

A (submesh_types, spatial_methods) tuple for PyBaMM usage.

ep_bolfi.models.standard_parameters

Comprehensive list of parameters of every model.

SI units are assumed unless stated otherwise. When imported, this package turns into the list of variables contained in here. The "syntactic sugar" with the lower and upper indexed letters is treated by Python to be the same as their normal counterparts. E.g., "a□" and "an" refer to the exact same variable. Greek letters aren't converted, unless they are also indexed, in which case the same as before applies: "ε^{β}" is the same as "εβ".

Function parameters are defined in accordance with PyBaMM and noted with their units in the code here, so for their documentation, look there.

Module Attributes

R	Gas constant.
F	Faraday constant.
k_B	Boltzmann constant.
qe	Electron volt.
T_ref	Reference temperature for non-dimensionalization.
T_init	Initial temperature.
Ln_dim	Anode thickness.
Ls_dim	Separator thickness.
Lp_dim	Cathode thickness.
L_dim	Cell thickness.
A	Cross-section area of the cell.
V	Volume of the cell.
L_x	PyBaMM-compatible name for the cell thickness.
L_y	Width of the electrode for PyBaMM compatibility.
L_z	Height of the electrode for PyBaMM compatibility.
C	C-rate of the battery (in A).
Ul	Lower threshold for the cell voltage.
Uu	Upper threshold for the cell voltage.
ce_typ	Reference electrolyte concentration for non-dimensionalization.
cn	Maximum lithium concentration in the anode active material.
cp	Maximum lithium concentration in the cathode active material.
an_dim	Specific surface area of the anode.
ap_dim	Specific surface area of the cathode.
Rn	Mean/Median radius of the anode particles.
Rp	Mean/Median radius of the cathode particles.
zn	Charge number of the anode interface reaction.
zp	Charge number of the cathode interface reaction.
zn_salt	Charge number of the salt dissociation (anion).
zp_salt	Charge number of the salt dissociation (cation).

continues on next page

Table 7 – continued from previous page

nn	Stoichiometric coefficient of the salt dissociation (anion).
np	Stoichiometric coefficient of the salt dissociation (cation).
ce_dim_init	Initial electrolyte concentration.
ce_init	Non-dimensionalized initial electrolyte concentration.
M_N	Solvent molar mass.
c_N	Solvent concentration.
v_N	Solvent partial molar volume.
tilde_ρ_N	Solvent partial mass density (i.e., weight change per volume).
C_DLn	Negative electrode double-layer capacity.
C_DLp	Positive electrode double-layer capacity.
L_SEI	Thickness of the SEI.
R_SEI	Resistance of the SEI.
permittivity_SEI	Relative permittivity of the SEI.
C_SEI	Capacitance of the SEI.
t_SEI_minus	Transference number of anions in the SEI (with reference to the center-of-mass velocity).
L_electrolyte_for_SEI_model	Length-scale for bulk electrolyte contribution in SEI model.
De_typ	Reference electrolyte diffusivity for non-dimensionalization.
Dn_typ	Reference anode diffusivity for non-dimensionalization.
Dp_typ	Reference cathode diffusivity for non-dimensionalization.
isen_0_ref	Reference anode exchange current density for non-dimensionalization.
isep_0_ref	Reference cathode exchange current density for non-dimensionalization.
OCVn_ref	Reference anode OCV for non-dimensionalization.
OCVp_ref	Reference cathode OCV for non-dimensionalization.
an	Non-dimensionalized specific surface area of the anode.
ap	Non-dimensionalized specific surface area of the cathode.
timescale	Choose the discharge timescale for non-dimensionalization.
Ce	Non-dimensionalized electrolyte diffusion timescale.
Cn	Non-dimensionalized anode diffusion timescale.
Cp	Non-dimensionalized cathode diffusion timescale.
Crn	Non-dimensionalized anode interface reaction timescale.
Crp	Non-dimensionalized cathode interface reaction timescale.
Ln	Non-dimensionalized anode thickness.
Ls	Non-dimensionlized separator thickness.
Lp	Non-dimensionalized cathode thickness.
Le	Non-dimensionalized thicknesses for the whole cell.
I_extern_dim	Externally applied current for galvanostatic operation (in A).
I_extern	Non-dimensionalized external current.
n_electrodes_parallel	Current divider.
n_cells	Voltage multiplier.
I_typ	Copy of the C-rate of the battery for PyBaMM compatibility.
A_cc	Copy of the cross-section area for PyBaMM compatibility.
current_with_time	Copy of the external current for PyBaMM compatibility.

continues on next page

Table 7 – continued from previous page

dimensional_current_with_time	Copy of the dimensional external current for PyBaMM compatibility.
dimensional_current_density_with_time	Copy of the dimensional current density for PyBaMM compatibility.
voltage_low_cut	Copy of the lower voltage threshold for PyBaMM compatibility.
voltage_high_cut	Copy of the upper voltage threshold for PyBaMM compatibility.
capacity	Cell capacity for calculating amperage from C-rates (in Ah).

ep_bolfi.models.standard_parameters.R

ep_bolfi.models.standard_parameters.**R = Scalar(0x394def62905509c8, 8.314462618, children=[], domains={})**

Gas constant.

ep_bolfi.models.standard_parameters.F

ep_bolfi.models.standard_parameters.**F = Scalar(-0x1e073b3326820740, 96485.33212, children=[], domains={})**

Faraday constant.

ep_bolfi.models.standard_parameters.k_B

ep_bolfi.models.standard_parameters.**k_B = 1.380649e-23**

Boltzmann constant.

ep_bolfi.models.standard_parameters.qe

ep_bolfi.models.standard_parameters.**qe = 1.602176634e-19**

Electron volt.

ep_bolfi.models.standard_parameters.T_ref

ep_bolfi.models.standard_parameters.**T_ref = Parameter(0x26443c261acad3dd, Reference temperature [K], children=[], domains={})**

Reference temperature for non-dimensionalization.

ep_bolfi.models.standard_parameters.T_init

ep_bolfi.models.standard_parameters.**T_init = Parameter(-0x6c3b6449c9331afe, Initial temperature [K], children=[], domains={})**

Initial temperature.

ep_bolfi.models.standard_parameters.Ln_dim

ep_bolfi.models.standard_parameters.**Ln_dim = Parameter(0x204ddcf74c8b25c, Negative electrode thickness [m], children=[], domains={})**

Anode thickness.

ep_bolfi.models.standard_parameters.Ls_dim

ep_bolfi.models.standard_parameters.**Ls_dim = Parameter(-0x675f2659ebed5a1b, Separator thickness [m], children=[], domains={})**

Separator thickness.

ep_bolfi.models.standard_parameters.Lp_dim

ep_bolfi.models.standard_parameters.**Lp_dim = Parameter(-0x46d49fb639290ba1, Positive electrode thickness [m], children=[], domains={})**

Cathode thickness.

ep_bolfi.models.standard_parameters.L_dim

ep_bolfi.models.standard_parameters.**L_dim = Addition(-0x53ba76eb8d5fc773, +, children=['Negative electrode thickness [m] + Separator thickness [m]', 'Positive electrode thickness [m]'], domains={})**

Cell thickness.

ep_bolfi.models.standard_parameters.A

ep_bolfi.models.standard_parameters.**A = Parameter(-0x4afa93d1dc92b1f4, Current collector perpendicular area [m2], children=[], domains={})**

Cross-section area of the cell.

ep_bolfi.models.standard_parameters.V

ep_bolfi.models.standard_parameters.**V = Parameter(-0x47695e98c9495f54, Cell volume [m3], children=[], domains={})**

Volume of the cell. Left as a separate Parameter for compatibility.

ep_bolfi.models.standard_parameters.L_x

ep_bolfi.models.standard_parameters.**L_x = Addition(-0x53ba76eb8d5fc773, +, children=['Negative electrode thickness [m] + Separator thickness [m]', 'Positive electrode thickness [m]'], domains={})**

PyBaMM-compatible name for the cell thickness.

ep_bolfi.models.standard_parameters.L_y

ep_bolfi.models.standard_parameters.**L_y = Parameter(0x3ba35f9658437148, Electrode width [m], children=[], domains={})**

Width of the electrode for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.L_z

ep_bolfi.models.standard_parameters.**L_z = Parameter(-0x6d4d4ea7ebab9b4c, Electrode height [m], children=[], domains={})**

Height of the electrode for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.C

ep_bolfi.models.standard_parameters.**C = Parameter(0x65b06ee96a4ac89c, Typical current [A], children=[], domains={})**

C-rate of the battery (in A).

ep_bolfi.models.standard_parameters.Ul

ep_bolfi.models.standard_parameters.**Ul = Parameter(-0x4875137260a8b526, Lower voltage cut-off [V], children=[], domains={})**

Lower threshold for the cell voltage.

ep_bolfi.models.standard_parameters.Uu

ep_bolfi.models.standard_parameters.**Uu = Parameter(-0xdd7fdeb0ae5e1fc, Upper voltage cut-off [V], children=[], domains={})**

Upper threshold for the cell voltage.

ep_bolfi.models.standard_parameters.ce_typ

ep_bolfi.models.standard_parameters.**ce_typ = Parameter(0x7f5ec946f6a93799, Typical electrolyte concentration [mol.m-3], children=[], domains={})**

Reference electrolyte concentration for non-dimensionalization.

ep_bolfi.models.standard_parameters.an_dim

ep_bolfi.models.standard_parameters.**an_dim = Parameter(-0x588a3973fb9d23d, Negative electrode surface area to volume ratio [m-1], children=[], domains={})**

Specific surface area of the anode.

ep_bolfi.models.standard_parameters.ap_dim

ep_bolfi.models.standard_parameters.**ap_dim = Parameter(0x784aaf7031d87e7a, Positive electrode surface area to volume ratio [m-1], children=[], domains={})**

Specific surface area of the cathode.

ep_bolfi.models.standard_parameters.Rn

ep_bolfi.models.standard_parameters.**Rn = Parameter(-0x5669516f734419e1, Negative particle radius [m], children=[], domains={})**

Mean/Median radius of the anode particles.

ep_bolfi.models.standard_parameters.Rp

ep_bolfi.models.standard_parameters.**Rp = Parameter(0x45968a75c48ec3ab, Positive particle radius [m], children=[], domains={})**

Mean/Median radius of the cathode particles.

ep_bolfi.models.standard_parameters.zn

ep_bolfi.models.standard_parameters.**zn = Parameter(0xad810ff61cb4d69, Negative electrode electrons in reaction, children=[], domains={})**

Charge number of the anode interface reaction.

ep_bolfi.models.standard_parameters.zp

ep_bolfi.models.standard_parameters.**zp = Parameter(0x6116135bd8fc4919, Positive electrode electrons in reaction, children=[], domains={})**

Charge number of the cathode interface reaction.

ep_bolfi.models.standard_parameters.zn_salt

ep_bolfi.models.standard_parameters.**zn_salt = Parameter(-0x70acdae5d9eee413, Charge number of anion in electrolyte salt dissociation, children=[], domains={})**

Charge number of the salt dissociation (anion).

ep_bolfi.models.standard_parameters.zp_salt

ep_bolfi.models.standard_parameters.**zp_salt = Parameter(-0x5e2400fb808807a, Charge number of cation in electrolyte salt dissociation, children=[], domains={})**

Charge number of the salt dissociation (cation).

ep_bolfi.models.standard_parameters.nn

ep_bolfi.models.standard_parameters.**nn = Parameter(-0x6500a98a9de13c5e, Stoichiometry of anion in electrolyte salt dissociation, children=[], domains={})**

Stoichiometric coefficient of the salt dissociation (anion).

ep_bolfi.models.standard_parameters.np

ep_bolfi.models.standard_parameters.**np = Parameter(0x1af56720fb049332, Stoichiometry of cation in electrolyte salt dissociation, children=[], domains={})**

Stoichiometric coefficient of the salt dissociation (cation).

ep_bolfi.models.standard_parameters.ce_dim_init

ep_bolfi.models.standard_parameters.**ce_dim_init = Parameter(0x529a3bf5197ac2fc, Initial concentration in electrolyte [mol.m-3], children=[], domains={})**

Initial electrolyte concentration.

ep_bolfi.models.standard_parameters.ce_init

ep_bolfi.models.standard_parameters.**ce_init = Division(-0x68e9ef479a0437f2, /, children=['Initial concentration in electrolyte [mol.m-3]', 'Typical electrolyte concentration [mol.m-3]'], domains={})**

Non-dimensionalized initial electrolyte concentration.

ep_bolfi.models.standard_parameters.M_N

ep_bolfi.models.standard_parameters.**M_N = Parameter(0x3ce62b1c9a181c2b, Molar mass of electrolyte solvent [kg.mol-1], children=[], domains={})**

Solvent molar mass.

ep_bolfi.models.standard_parameters.c_N

ep_bolfi.models.standard_parameters.**c_N = Parameter(-0x7200223b9297055a, Solvent concentration [mol.m-3], children=[], domains={})**

Solvent concentration.

ep_bolfi.models.standard_parameters.v_N

ep_bolfi.models.standard_parameters.**v_N = Parameter(0x3415d9e9c5cd8a53, Partial molar volume of electrolyte solvent [m3.mol-1], children=[], domains={})**

Solvent partial molar volume.

ep_bolfi.models.standard_parameters.C_DLn

ep_bolfi.models.standard_parameters.**C_DLn = Parameter(-0x744e1bc5827ed78a, Negative electrode double-layer capacity [F.m-2], children=[], domains={})**

Negative electrode double-layer capacity.

ep_bolfi.models.standard_parameters.C_DLp

ep_bolfi.models.standard_parameters.**C_DLp = Parameter(-0x4164b3f81e4c61c1, Positive electrode double-layer capacity [F.m-2], children=[], domains={})**

Positive electrode double-layer capacity.

ep_bolfi.models.standard_parameters.L_SEI

ep_bolfi.models.standard_parameters.**L_SEI = Parameter(-0x6bf37276b3196e15, SEI thickness [m], children=[], domains={})**

Thickness of the SEI.

ep_bolfi.models.standard_parameters.R_SEI

ep_bolfi.models.standard_parameters.**R_SEI = Division(0x7b00e92e68e706ee, /, children=['SEI thickness [m]', 'SEI ionic conductivity [S.m-1]'], domains={})**

Resistance of the SEI.

ep_bolfi.models.standard_parameters.permittivity_SEI

ep_bolfi.models.standard_parameters.**permittivity_SEI = Parameter(-0x4f9f3b84ae127948, SEI relative permittivity, children=[], domains={})**

Relative permittivity of the SEI.

ep_bolfi.models.standard_parameters.C_SEI

ep_bolfi.models.standard_parameters.**C_SEI = Division(0x530ac0fe2c9d757a, /, children=['8.8541878128e-12 * SEI relative permittivity', 'SEI thickness [m]'], domains={})**

Capacitance of the SEI.

ep_bolfi.models.standard_parameters.t_SEI_minus

ep_bolfi.models.standard_parameters.**t_SEI_minus = Parameter(-0x2f4ee5d65e80347a, Anion transference number in SEI, children=[], domains={})**

Transference number of anions in the SEI (with reference to the center-of-mass velocity).

ep_bolfi.models.standard_parameters.L_electrolyte_for_SEI_model

ep_bolfi.models.standard_parameters.**L_electrolyte_for_SEI_model = Addition(0x257539d83ad61bbf, +, children=['0.25 * (Negative electrode thickness [m] + Positive electrode thickness [m])', '0.5 * Separator thickness [m]'], domains={})**

Length-scale for bulk electrolyte contribution in SEI model.

ep_bolfi.models.standard_parameters.De_typ

ep_bolfi.models.standard_parameters.**De_typ =**
FunctionParameter(0x3043264f2c9015c6, Electrolyte diffusivity [m2.s-1], children=['Typical electrolyte concentration [mol.m-3]', 'Reference temperature [K]'], domains={})

Reference electrolyte diffusivity for non-dimensionalization.

ep_bolfi.models.standard_parameters.Dn_typ

ep_bolfi.models.standard_parameters.**Dn_typ =**
FunctionParameter(0x1dfc29ecf727645a, Negative particle diffusivity [m2.s-1], children=['Initial concentration in negative electrode [mol.m-3] / Maximum concentration in negative electrode [mol.m-3]', 'Reference temperature [K]'], domains={})

Reference anode diffusivity for non-dimensionalization.

ep_bolfi.models.standard_parameters.Dp_typ

ep_bolfi.models.standard_parameters.**Dp_typ =**
FunctionParameter(-0xc805a6a32ca8e4b, Positive particle diffusivity [m2.s-1], children=['Initial concentration in positive electrode [mol.m-3] / Maximum concentration in positive electrode [mol.m-3]', 'Reference temperature [K]'], domains={})

Reference cathode diffusivity for non-dimensionalization.

ep_bolfi.models.standard_parameters.isen_0_ref

ep_bolfi.models.standard_parameters.**isen_0_ref =**
FunctionParameter(0x2cb5eede75617a50, Negative electrode exchange-current density [A.m-2], children=['Typical electrolyte concentration [mol.m-3]', 'Initial concentration in negative electrode [mol.m-3]', 'Maximum concentration in negative electrode [mol.m-3]', 'Reference temperature [K]'], domains={})

Reference anode exchange current density for non-dimensionalization.

ep_bolfi.models.standard_parameters.isep_0_ref

ep_bolfi.models.standard_parameters.**isep_0_ref =**
FunctionParameter(0x4ebec0e3f7651b1, Positive electrode exchange-current density [A.m-2], children=['Typical electrolyte concentration [mol.m-3]', 'Initial concentration in positive electrode [mol.m-3]', 'Maximum concentration in positive electrode [mol.m-3]', 'Reference temperature [K]'], domains={})

Reference cathode exchange current density for non-dimensionalization.

ep_bolfi.models.standard_parameters.OCVn_ref

ep_bolfi.models.standard_parameters.**OCVn_ref = FunctionParameter(-0x29d7f50c078908b, Negative electrode OCP [V], children=['Initial concentration in negative electrode [mol.m-3] / Maximum concentration in negative electrode [mol.m-3]'], domains={})**

Reference anode OCV for non-dimensionalization.

ep_bolfi.models.standard_parameters.OCVp_ref

ep_bolfi.models.standard_parameters.**OCVp_ref = FunctionParameter(-0x190bf43083a05f8e, Positive electrode OCP [V], children=['Initial concentration in positive electrode [mol.m-3] / Maximum concentration in positive electrode [mol.m-3]'], domains={})**

Reference cathode OCV for non-dimensionalization.

ep_bolfi.models.standard_parameters.an

ep_bolfi.models.standard_parameters.**an = Multiplication(-0x2495890628675ce9, *, children=['Negative electrode surface area to volume ratio [m-1]', 'Negative particle radius [m]'], domains={})**

Non-dimensionalized specific surface area of the anode.

ep_bolfi.models.standard_parameters.ap

ep_bolfi.models.standard_parameters.**ap = Multiplication(0x3c420db5f3e8362, *, children=['Positive electrode surface area to volume ratio [m-1]', 'Positive particle radius [m]'], domains={})**

Non-dimensionalized specific surface area of the cathode.

ep_bolfi.models.standard_parameters.timescale

ep_bolfi.models.standard_parameters.**timescale = Division(-0x79467c95c3db211, /, children=['96485.33212 * Maximum concentration in positive electrode [mol.m-3] * (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m])', 'Typical current [A] / Current collector perpendicular area [m2]'], domains={})**

Choose the discharge timescale for non-dimensionalization.

ep_bolfi.models.standard_parameters.Ce

ep_bolfi.models.standard_parameters.**Ce = Division(-0x3f4b0597d3e34958, /, children=['((Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]) ** 2.0) / Electrolyte diffusivity [m2.s-1]', '96485.33212 * Maximum concentration in positive electrode [mol.m-3] * (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]) / (Typical current [A] / Current collector perpendicular area [m2])'], domains={})**

Non-dimensionalized electrolyte diffusion timescale.

ep_bolfi.models.standard_parameters.cn

ep_bolfi.models.standard_parameters.**cn = Parameter(-0xd09eb3556d06f13, Maximum concentration in negative electrode [mol.m-3], children=[], domains={})**

Maximum lithium concentration in the anode active material.

ep_bolfi.models.standard_parameters.cp

ep_bolfi.models.standard_parameters.**cp = Parameter(0x103031490cfae5e4, Maximum concentration in positive electrode [mol.m-3], children=[], domains={})**

Maximum lithium concentration in the cathode active material.

ep_bolfi.models.standard_parameters.Crn

ep_bolfi.models.standard_parameters.**Crn = Division(0x3c6303cbc4d7622a, /, children=['96485.33212 * Maximum concentration in negative electrode [mol.m-3] / (Negative electrode exchange-current density [A.m-2] * Negative electrode surface area to volume ratio [m-1])', '96485.33212 * Maximum concentration in positive electrode [mol.m-3] * (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]) / (Typical current [A] / Current collector perpendicular area [m2])'], domains={})**

Non-dimensionalized anode interface reaction timescale.

ep_bolfi.models.standard_parameters.Crp

ep_bolfi.models.standard_parameters.**Crp = Division(0x7c252e3ecd948199, /, children=['96485.33212 * Maximum concentration in positive electrode [mol.m-3] / (Positive electrode exchange-current density [A.m-2] * Positive electrode surface area to volume ratio [m-1])', '96485.33212 * Maximum concentration in positive electrode [mol.m-3] * (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]) / (Typical current [A] / Current collector perpendicular area [m2])'], domains={})**

Non-dimensionalized cathode interface reaction timescale.

ep_bolfi.models.standard_parameters.Ln

ep_bolfi.models.standard_parameters.**Ln = Division(0x54547d27bf91c0ed, /, children=['Negative electrode thickness [m]', 'Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]'], domains={})**

Non-dimensionalized anode thickness.

ep_bolfi.models.standard_parameters.Ls

ep_bolfi.models.standard_parameters.**Ls = Division(0x4a3830b997f6c6ac, /, children=['Separator thickness [m]', 'Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]'], domains={})**

Non-dimensionlized separator thickness.

ep_bolfi.models.standard_parameters.Lp

ep_bolfi.models.standard_parameters.**Lp = Division(-0x92a6af1688d8e38, /, children=['Positive electrode thickness [m]', 'Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]'], domains={})**

Non-dimensionalized cathode thickness.

ep_bolfi.models.standard_parameters.Le

ep_bolfi.models.standard_parameters.**Le = Concatenation(0x32bea0cc3daef050, concatenation, children=['broadcast(Negative electrode thickness [m] / (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]))', 'broadcast(Separator thickness [m] / (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]))', 'broadcast(Positive electrode thickness [m] / (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]))'], domains={'primary': ['negative electrode', 'separator', 'positive electrode']})**

Non-dimensionalized thicknesses for the whole cell.

ep_bolfi.models.standard_parameters.I_extern_dim

ep_bolfi.models.standard_parameters.**I_extern_dim = FunctionParameter(0x5d4252ea3dc2046c, Current function [A], children=['time * 96485.33212 * Maximum concentration in positive electrode [mol.m-3] * (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]) / (Typical current [A] / Current collector perpendicular area [m2])'], domains={})**

Externally applied current for galvanostatic operation (in A). Please note that the variable name is important for PyBaMM comp..

ep_bolfi.models.standard_parameters.I_extern

ep_bolfi.models.standard_parameters.**I_extern = Division(-0x3b540dfe4dfbbbfa, /, children=['Current function [A]', 'Typical current [A]'], domains={})**

Non-dimensionalized external current.

ep_bolfi.models.standard_parameters.n_electrodes_parallel

ep_bolfi.models.standard_parameters.**n_electrodes_parallel = Parameter(0x67e8966202552ccd, Number of electrodes connected in parallel to make a cell, children=[], domains={})**

Current divider.

ep_bolfi.models.standard_parameters.n_cells

ep_bolfi.models.standard_parameters.**n_cells = Parameter(0x37bc39b99b9103a3, Number of cells connected in series to make a battery, children=[], domains={})**

Voltage multiplier.

ep_bolfi.models.standard_parameters.I_typ

ep_bolfi.models.standard_parameters.**I_typ = Parameter(0x65b06ee96a4ac89c, Typical current [A], children=[], domains={})**

Copy of the C-rate of the battery for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.A_cc

ep_bolfi.models.standard_parameters.**A_cc = Parameter(-0x4afa93d1dc92b1f4, Current collector perpendicular area [m2], children=[], domains={})**

Copy of the cross-section area for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.current_with_time

ep_bolfi.models.standard_parameters.**current_with_time = Division(-0x3b540dfe4dfbbbfa, /, children=['Current function [A]', 'Typical current [A]'], domains={})**

Copy of the external current for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.dimensional_current_with_time

ep_bolfi.models.standard_parameters.**dimensional_current_with_time = FunctionParameter(0x5d4252ea3dc2046c, Current function [A], children=['time * 96485.33212 * Maximum concentration in positive electrode [mol.m-3] * (Negative electrode thickness [m] + Separator thickness [m] + Positive electrode thickness [m]) / (Typical current [A] / Current collector perpendicular area [m2])'], domains={})**

Copy of the dimensional external current for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.dimensional_current_density_with_time

ep_bolfi.models.standard_parameters.**dimensional_current_density_with_time = Division(0x3f969a1ec972a37e, /, children=['Current function [A]', 'Current collector perpendicular area [m2]'], domains={})**

Copy of the dimensional current density for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.voltage_low_cut

ep_bolfi.models.standard_parameters.**voltage_low_cut = Parameter(-0x4875137260a8b526, Lower voltage cut-off [V], children=[], domains={})**

Copy of the lower voltage threshold for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.voltage_high_cut

ep_bolfi.models.standard_parameters.**voltage_high_cut = Parameter(-0xdd7fdeb0ae5e1fc, Upper voltage cut-off [V], children=[], domains={})**

Copy of the upper voltage threshold for PyBaMM compatibility.

ep_bolfi.models.standard_parameters.capacity

ep_bolfi.models.standard_parameters.**capacity = Parameter(0x1f8fbd6212d32cfb, Nominal cell capacity [A.h], children=[], domains={})**

Cell capacity for calculating amperage from C-rates (in Ah).

Functions

De(ce, T)	Non-dimensionalized electrolyte diffusivity.
De_dim(ce_dim, T_dim)	Electrolyte diffusivity.
Dn(SOCn, T)	Non-dimensionalized anode diffusivity.
Dn_dim(SOCn, T_dim)	Anode diffusivity.
Dp(SOCp, T)	Non-dimensionalized cathode diffusivity.
Dp_dim(SOCp, T_dim)	Cathode diffusivity.

continues on next page

Table 8 – continued from previous page

OCVn(SOCn, T)	Non-dimensionalized anode OCV.
OCVn_dim(SOCn, T_dim)	Anode OCV.
OCVp(SOCp, T)	Non-dimensionalized cathode OCV.
OCVp_dim(SOCp, T_dim)	Cathode OCV.
SOCn_dim_init(x)	Initial SOC of the anode.
SOCn_init(x)	Non-dimensionalized initial SOC of the anode.
SOCp_dim_init(x)	Initial SOC of the cathode.
SOCp_init(x)	Non-dimensionalized initial SOC of the cathode.
dOCVn_dSOCn(SOCn, T)	Non-dimensionalized ∂ anode OCV / ∂ anode SOC.
dOCVn_dT(SOCn)	Non-dimensionalized ∂ anode OCV / ∂ temperature.
dOCVn_dT_dSOCn(SOCn)	Non-dimensionalized (∂ anode OCV / ∂ temperature) / ∂ anode SOC.
dOCVn_dT_dSOCn_dim(SOCn)	(∂ anode OCV / ∂ temperature) / ∂ anode SOC.
dOCVn_dT_dim(SOCn)	∂ anode OCV / ∂ temperature.
dOCVn_dim_dSOCn(SOCn, T_dim)	∂ anode OCV / ∂ anode SOC.
dOCVp_dSOCp(SOCp, T)	Non-dimensionalized ∂ cathode OCV / ∂ cathode SOC.
dOCVp_dT(SOCp)	Non-dimensionalized ∂ cathode OCV / ∂ temperature.
dOCVp_dT_dSOCp(SOCp)	Non-dimensionalized (∂ cathode OCV / ∂ temperature) / ∂ cathode SOC.
dOCVp_dT_dSOCp_dim(SOCp)	(∂ cathode OCV / ∂ temperature) / ∂ cathode SOC.
dOCVp_dT_dim(SOCp)	∂ cathode OCV / ∂ temperature.
dOCVp_dim_dSOCp(SOCp, T_dim)	∂ cathode OCV / ∂ cathode SOC.
d_cen_isen_0(cen, SOCn_surf, cn_max, T)	The non-dimensionalized version of the prior variable.
d_cen_isen_0_dim(cen_dim, SOCn_surf_dim, ...)	∂ anode exchange current density / ∂ electrolyte concentration.
d_cep_isep_0(cep, SOCp_surf, cp_max, T)	The non-dimensionalized version of the prior variable.
d_cep_isep_0_dim(cep_dim, SOCp_surf_dim, ...)	∂ cathode exchange current density / ∂ electrolyte concentration.
isen_0(cen, SOCn_surf, cn_max, T)	Non-dimensionalized anode exchange current density.
isen_0_dim(cen_dim, SOCn_surf_dim, cn_max, T_dim)	Anode exchange current density.
isep_0(cep, SOCp_surf, cp_max, T)	Non-dimensionalized cathode exchange current density.
isep_0_dim(cep_dim, SOCp_surf_dim, cp_max, T_dim)	Cathode exchange current density.
one_plus_dlnf_dlnc(ce)	Non-dimensionalized (referring to the input) thermodynamic factor.
one_plus_dlnf_dlnc_dim(ce_dim)	! Thermodynamic factor.
t_plus(ce)	Non-dimensionalized (referring to the input) transference number.
t_plus_dim(ce_dim)	Transference number.

ep_bolfi.models.standard_parameters.De

ep_bolfi.models.standard_parameters.**De**(*ce*, *T*)

Non-dimensionalized electrolyte diffusivity.

ep_bolfi.models.standard_parameters.De_dim

ep_bolfi.models.standard_parameters.**De_dim**(*ce_dim*, *T_dim*)

Electrolyte diffusivity.

ep_bolfi.models.standard_parameters.Dn

ep_bolfi.models.standard_parameters.**Dn**(*SOCn*, *T*)

Non-dimensionalized anode diffusivity.

ep_bolfi.models.standard_parameters.Dn_dim

ep_bolfi.models.standard_parameters.**Dn_dim**(*SOCn*, *T_dim*)

Anode diffusivity.

ep_bolfi.models.standard_parameters.Dp

ep_bolfi.models.standard_parameters.**Dp**(*SOCp*, *T*)

Non-dimensionalized cathode diffusivity.

ep_bolfi.models.standard_parameters.Dp_dim

ep_bolfi.models.standard_parameters.**Dp_dim**(*SOCp*, *T_dim*)

Cathode diffusivity.

ep_bolfi.models.standard_parameters.OCVn

ep_bolfi.models.standard_parameters.**OCVn**(*SOCn*, *T*)

Non-dimensionalized anode OCV.

ep_bolfi.models.standard_parameters.OCVn_dim

ep_bolfi.models.standard_parameters.**OCVn_dim**(*SOCn*, *T_dim*)

Anode OCV.

ep_bolfi.models.standard_parameters.OCVp

ep_bolfi.models.standard_parameters.**OCVp**(*SOCp*, *T*)

Non-dimensionalized cathode OCV.

ep_bolfi.models.standard_parameters.OCVp_dim

ep_bolfi.models.standard_parameters.**OCVp_dim**(*SOCp*, *T_dim*)

Cathode OCV.

ep_bolfi.models.standard_parameters.SOCn_dim_init

ep_bolfi.models.standard_parameters.**SOCn_dim_init**(*x*)

Initial SOC of the anode.

ep_bolfi.models.standard_parameters.SOCn_init

ep_bolfi.models.standard_parameters.**SOCn_init**(*x*)

Non-dimensionalized initial SOC of the anode.

ep_bolfi.models.standard_parameters.SOCp_dim_init

ep_bolfi.models.standard_parameters.**SOCp_dim_init**(*x*)

Initial SOC of the cathode.

ep_bolfi.models.standard_parameters.SOCp_init

ep_bolfi.models.standard_parameters.**SOCp_init**(*x*)

Non-dimensionalized initial SOC of the cathode.

ep_bolfi.models.standard_parameters.dOCVn_dSOCn

ep_bolfi.models.standard_parameters.**dOCVn_dSOCn**(*SOCn*, *T*)

Non-dimensionalized ∂ anode OCV / ∂ anode SOC.

ep_bolfi.models.standard_parameters.dOCVn_dT

ep_bolfi.models.standard_parameters.**dOCVn_dT**(*SOCn*)

Non-dimensionalized ∂ anode OCV / ∂ temperature.

ep_bolfi.models.standard_parameters.dOCVn_dT_dSOCn

ep_bolfi.models.standard_parameters.**dOCVn_dT_dSOCn**(*SOCn*)
Non-dimensionalized (∂ anode OCV / ∂ temperature) / ∂ anode SOC.

ep_bolfi.models.standard_parameters.dOCVn_dT_dSOCn_dim

ep_bolfi.models.standard_parameters.**dOCVn_dT_dSOCn_dim**(*SOCn*)
(∂ anode OCV / ∂ temperature) / ∂ anode SOC.

ep_bolfi.models.standard_parameters.dOCVn_dT_dim

ep_bolfi.models.standard_parameters.**dOCVn_dT_dim**(*SOCn*)
∂ anode OCV / ∂ temperature.

ep_bolfi.models.standard_parameters.dOCVn_dim_dSOCn

ep_bolfi.models.standard_parameters.**dOCVn_dim_dSOCn**(*SOCn*, *T_dim*)
∂ anode OCV / ∂ anode SOC.

ep_bolfi.models.standard_parameters.dOCVp_dSOCp

ep_bolfi.models.standard_parameters.**dOCVp_dSOCp**(*SOCp*, *T*)
Non-dimensionalized ∂ cathode OCV / ∂ cathode SOC.

ep_bolfi.models.standard_parameters.dOCVp_dT

ep_bolfi.models.standard_parameters.**dOCVp_dT**(*SOCp*)
Non-dimensionalized ∂ cathode OCV / ∂ temperature.

ep_bolfi.models.standard_parameters.dOCVp_dT_dSOCp

ep_bolfi.models.standard_parameters.**dOCVp_dT_dSOCp**(*SOCp*)
Non-dimensionalized (∂ cathode OCV / ∂ temperature) / ∂ cathode SOC.

ep_bolfi.models.standard_parameters.dOCVp_dT_dSOCp_dim

ep_bolfi.models.standard_parameters.**dOCVp_dT_dSOCp_dim**(*SOCp*)
(∂ cathode OCV / ∂ temperature) / ∂ cathode SOC.

ep_bolfi.models.standard_parameters.dOCVp_dT_dim

ep_bolfi.models.standard_parameters.**dOCVp_dT_dim**(*SOCp*)

∂ cathode OCV / ∂ temperature.

ep_bolfi.models.standard_parameters.dOCVp_dim_dSOCp

ep_bolfi.models.standard_parameters.**dOCVp_dim_dSOCp**(*SOCp*, *T_dim*)

∂ cathode OCV / ∂ cathode SOC.

ep_bolfi.models.standard_parameters.d_cen_isen_0

ep_bolfi.models.standard_parameters.**d_cen_isen_0**(*cen*, *SOCn_surf*, *cn_max*, *T*)

The non-dimensionalized version of the prior variable.

ep_bolfi.models.standard_parameters.d_cen_isen_0_dim

ep_bolfi.models.standard_parameters.**d_cen_isen_0_dim**(*cen_dim*, *SOCn_surf_dim*, *cn_max*, *T_dim*)

∂ anode exchange current density / ∂ electrolyte concentration.

ep_bolfi.models.standard_parameters.d_cep_isep_0

ep_bolfi.models.standard_parameters.**d_cep_isep_0**(*cep*, *SOCp_surf*, *cp_max*, *T*)

The non-dimensionalized version of the prior variable.

ep_bolfi.models.standard_parameters.d_cep_isep_0_dim

ep_bolfi.models.standard_parameters.**d_cep_isep_0_dim**(*cep_dim*, *SOCp_surf_dim*, *cp_max*, *T_dim*)

∂ cathode exchange current density / ∂ electrolyte concentration.

ep_bolfi.models.standard_parameters.isen_0

ep_bolfi.models.standard_parameters.**isen_0**(*cen*, *SOCn_surf*, *cn_max*, *T*)

Non-dimensionalized anode exchange current density.

ep_bolfi.models.standard_parameters.isen_0_dim

ep_bolfi.models.standard_parameters.**isen_0_dim**(*cen_dim*, *SOCn_surf_dim*, *cn_max*, *T_dim*)

Anode exchange current density.

ep_bolfi.models.standard_parameters.isep_0

ep_bolfi.models.standard_parameters.**isep_0**(*cep*, *SOCp_surf*, *cp_max*, *T*)

Non-dimensionalized cathode exchange current density.

ep_bolfi.models.standard_parameters.isep_0_dim

ep_bolfi.models.standard_parameters.**isep_0_dim**(*cep_dim*, *SOCp_surf_dim*, *cp_max*, *T_dim*)

Cathode exchange current density.

ep_bolfi.models.standard_parameters.one_plus_dlnf_dlnc

ep_bolfi.models.standard_parameters.**one_plus_dlnf_dlnc**(*ce*)

Non-dimensionalized (referring to the input) thermodynamic factor.

ep_bolfi.models.standard_parameters.one_plus_dlnf_dlnc_dim

ep_bolfi.models.standard_parameters.**one_plus_dlnf_dlnc_dim**(*ce_dim*)

! Thermodynamic factor.

ep_bolfi.models.standard_parameters.t_plus

ep_bolfi.models.standard_parameters.**t_plus**(*ce*)

Non-dimensionalized (referring to the input) transference number.

ep_bolfi.models.standard_parameters.t_plus_dim

ep_bolfi.models.standard_parameters.**t_plus_dim**(*ce_dim*)

Transference number.

2.1.3 ep_bolfi.optimization

Modules

EP_BOLFI	This file contains functions to perform Expectation Propagation on simulator models using BOLFI (Bayesian Optimization for Likelihood- Free Inference).

ep_bolfi.optimization.EP_BOLFI

This file contains functions to perform Expectation Propagation on simulator models using BOLFI (Bayesian Optimization for Likelihood- Free Inference).

Functions

combine_parameters_to_try(parameters, ...)	Give every combination as full parameter sets.
fix_parameters(parameters_to_be_fixed)	Returns a function which sets some parameters in advance.

ep_bolfi.optimization.EP_BOLFI.combine_parameters_to_try

ep_bolfi.optimization.EP_BOLFI.**combine_parameters_to_try**(*parameters*, *parameters_to_try_dict*)

Give every combination as full parameter sets.

Parameters

- **parameters** – The base full parameter set as a dictionary.
- **parameters_to_try_dict** – The keys of this dictionary correspond to the *parameters*' keys where different values are to be inserted. These are given by the tuples which are the values of this dictionary.

Returns

A 2-tuple where the first item is the list of all parameter set combinations and the second the list of the combinations only.

ep_bolfi.optimization.EP_BOLFI.fix_parameters

ep_bolfi.optimization.EP_BOLFI.**fix_parameters**(*parameters_to_be_fixed*)

Returns a function which sets some parameters in advance.

Parameters

parameters_to_be_fixed – These parameters will at least be a part of the dictionary that the returned function returns.

Returns

The function which adds additional parameters to a dictionary or replaces existing parameters with the new ones.

Classes

EP_BOLFI(simulators, experimental_datasets, ...)	Expectation Propagation and Bayesian Optimization.
NDArrayEncoder(*[, skipkeys, ensure_ascii, ...])	Use with the JSON library to store NumPy arrays.
Optimizer_State(input_dim, mcmc_chains, ...)	Handles the heuristics for the EP-BOLFI operation modes.
Preprocessed_Simulator(simulator, ...[, ...])	Normalizes sampling to a standard normal distribution.

ep_bolfi.optimization.EP_BOLFI.EP_BOLFI

class ep_bolfi.optimization.EP_BOLFI.**EP_BOLFI**(*simulators*, *experimental_datasets*, *feature_extractors*, *fixed_parameters*, *free_parameters=None*, *initial_covariance=None*, *free_parameters_boundaries=None*, *boundaries_in_deviations=0*, *Q=None*, *r=None*, *Q_features=None*, *r_features=None*, *transform_parameters={}*, *weights=None*, *display_current_feature=None*, *fixed_parameter_order=None*)

Bases: `object`

Expectation Propagation and Bayesian Optimization.

Sets up and runs these two algorithms to infer model parameters. Use the variables *Q*, *r*, *Q_features* and *r_features* to copy the state of another estimator. Do not use them in any other case. Always use either all of them or none of them.

__init__(*simulators*, *experimental_datasets*, *feature_extractors*, *fixed_parameters*, *free_parameters=None*, *initial_covariance=None*, *free_parameters_boundaries=None*, *boundaries_in_deviations=0*, *Q=None*, *r=None*, *Q_features=None*, *r_features=None*, *transform_parameters={}*, *weights=None*, *display_current_feature=None*, *fixed_parameter_order=None*)

Parameters

- **simulators** – A list of functions that take one argument: a dictionary of combined *fixed_parameters* and *free_parameters*. They return the simulated counterpart to the experimental data. Most of the time, one function will be sufficient. Additional functions may be used to combine simulators which each give a subset of the total experimental method.
- **experimental_datasets** – A list of the experimental data. Each entry corresponds to the simulator in *simulators* with the same index and has the same structure as its output.
- **feature_extractors** – A list of functions which each take the corresponding data entry and return a list of numbers, which represent its reduced features.
- **fixed_parameters** – Dictionary of parameters that stay fixed and their values.
- **free_parameters** – Dictionary of parameters which shall be inferred and their initial guesses or, more accurately, their expected values. Please note that these values live in the transformed space. Optionally, the values may be a 2-tuple where the second entry would be the variance of that parameter. For finer tuning with covariances, use *initial_covariance* (will take precedence). Alternatively, you may set *free_parameters_boundaries* to set the expectation values and variances by confidence intervals.
- **initial_covariance** – Initial covariance of the parameters. Has to be a symmetric matrix (list of list or numpy 2D array). A reasonable simple choice is to have a diagonal matrix and set the standard deviation σ_i of each parameter to half of the distance between initial guess and biggest/smallest value that shall be tried. If the diagonal entries are σ_i^2 and the bounds are symmetric, the probability distribution of each parameter is 95% within these bounds. Please note that the same does not hold for the whole probability distribution.
- **free_parameters_boundaries** – Optional hard boundaries of the space in which optimal parameters are searched for. They are given as a dictionary with values as 2-tuples with the left and right boundaries. Boundaries need to be given for either none or all parameters. If None are given, boundaries will be set by *boundaries_in_deviations* relative to the covariance. The default then is the 95 % confidence ellipsoid. If neither *initial_covariance*

nor *free_parameters* set the covariance, this parameter sets it according to the example in the description of *initial_covariance*.

- **boundaries_in_deviations** – When <= 0, the boundaries are set as described above. When > 0, the boundaries are this multiple of the standard deviation. This scales with the shrinking covariance as the algorithm progresses. *free_parameters_boundaries* takes precedence, i.e., the covariance gets set and then the boundaries in the optimization are in standard deviations.
- **Q** – If you want to restore a previous EP-BOLFI instance from a dump of its data (see "result_to_json" method), put the *Q* attribute stored therein into this parameter.
- **r** – Same as *Q*, but use the *r* attribute.
- **Q_features** – Same as *Q*, but use the *Q_features* attribute.
- **r_features** – Same as *Q*, but use the *r_features* attribute.
- **transform_parameters** – Optional transformations between the parameter space that is used for searching for optimal parameters and the model parameters. Any missing free parameter is not transformed. The values are 2-tuples. The first entry is a function taking the search space parameter and returning the model parameter. The second entry is the inverse function. For convenience, any value may also be one of the following: - 'none' => (identity, identity) - 'log' => (exp, log) Please note that, for performance reasons, the returned inferred values are directly back-transformed from the mean of the internal standard distribution. This means that they represent the median of the actual distribution.
- **weights** – Optional weights to rescale multi-dimensional features. Has no effect on scalar features, as BOLFI is invariant with respect to constant or linear transformations. A list of lists of numpy.array which correspond to the *feature_extractors*. The numpy.array have to have the same length as their feature, and will be multiplied entry-wise onto the feature before taking the distance to the data.
- **display_current_feature** – A list of functions. Each corresponds to a feature extractor with the same index. Given an index of the array of its features, this returns a short description of it. If None is given, only the index will be shown in the output.
- **fixed_parameter_order** – Establish a numerical order to the parameter names. This prevents errors arising from internal reordering of the dictionaries. Only necessary when using the same model in different contexts.

Methods

__init__(simulators, experimental_datasets, ...)	
log_to_json()	Formats the relevant optimizer logs in JSON.
result_to_json([seed])	Formats the relevant optimizer states in JSON.
run([bolfi_initial_evidence, ...])	Runs Expectation Propagation together with BOLFI.
visualize_parameter_distribution()	Plots the features and visualizes the correlation.

Attributes

fixed_parameter_order	If the parameter order is explicitly given, use that.
log_of_tried_parameters	Stores all parameter combinations that have been tried.
experimental_features	Experimental features.
input_dim	Input dimension of the estimation task.
output_dim	Output dimension of the estimation task (sum of features).
simulator_index_by_feature	Mapping of index by all features to corresponding simulator.
sub_index_by_feature	Mapping of index by all features to that by one set of them.
weights	Set the weights to unity if None are given.
initial_guesses	Container for the initial expectation values.
log_of_raw_tried_parameters	Stores all raw parameter evaluation points.
log_of_discrepancies	Stores all discrepancies of the sampled parameters.
final_expectation	Stores the inference mean (empty at first).
final_covariance	Stores the inference covariance (empty at first).
final_correlation	Stores the inference correlation (empty at first).
initial_Q	Expectation Propagation covariance matrix (prior).
Q	Expectation Propagation covariance matrix (posterior).
initial_r	Expectation Propagation expectation value (prior).
r	Expectation Propagation expectation value (posterior).
Q_features	Expectation Propagation itemized covariance matrices.
r_features	Expectation Propagation itemized expectation values.
inferred_parameters	The inferred model parameters.
final_error_bounds	The 95% confidence bounds (which don't reflect the cross-correlations, but are easier to interpret).

Q

Expectation Propagation covariance matrix (posterior).

Q_features

Expectation Propagation itemized covariance matrices.

experimental_features

Experimental features.

final_correlation

Stores the inference correlation (empty at first).

final_covariance

Stores the inference covariance (empty at first).

final_error_bounds

The 95% confidence bounds (which don't reflect the cross-correlations, but are easier to interpret).

final_expectation

Stores the inference mean (empty at first).

fixed_parameter_order

If the parameter order is explicitly given, use that.

inferred_parameters

The inferred model parameters.

initial_Q

Expectation Propagation covariance matrix (prior).

initial_guesses

Container for the initial expectation values.

initial_r

Expectation Propagation expectation value (prior).

input_dim

Input dimension of the estimation task.

log_of_discrepancies

Stores all discrepancies of the sampled parameters.

log_of_raw_tried_parameters

Stores all raw parameter evaluation points.

log_of_tried_parameters

Stores all parameter combinations that have been tried.

log_to_json()

Formats the relevant optimizer logs in JSON.

output_dim

Output dimension of the estimation task (sum of features).

r

Expectation Propagation expectation value (posterior).

r_features

Expectation Propagation itemized expectation values.

result_to_json(*seed=None*)

Formats the relevant optimizer states in JSON.

Parameters

seed – Optionally put the seed you used when running EP-BOLFI.

run(*bolfi_initial_evidence=None*, *bolfi_total_evidence=None*, *bolfi_posterior_samples=None*, *ep_iterations=3*, *ep_dampener=None*, *final_dampening=None*, *ep_dampener_reduction_steps=-1*, *gelman_rubin_threshold=None*, *ess_ratio_resample=5.0*, *ess_ratio_sampling_from_zero=-1.0*, *ess_ratio_abort=20.0*, *max_heuristic_steps=10*, *posterior_sampling_increase=1.2*, *model_resampling_increase=1.1*, *independent_mcmc_chains=4*, *scramble_ep_feature_order=True*, *show_trials=False*, *verbose=True*, *seed=None*)

Runs Expectation Propagation together with BOLFI.

This function can be called multiple times; the estimation will take off from where it last stopped.

Enable parallelization in ELFI (for details, see https:// # elfi.readthedocs.io/en/latest/usage/parallelization.html): - For local multithreading with all cores:

```
elfi.set_client('multiprocessing')
```

- For scaling to clusters: `elfi.set_client('ipyparallel')`

Parameters

- **bolfi_initial_evidence** – Number of evidence samples BOLFI will take for each feature before using Bayesian Optimization sampling. Default: `1 + 2 ** number of estimated parameters`.
- **bolfi_total_evidence** – Number of evidence samples BOLFI will take for each feature in total (including initial evidence). Default: `2 * bolfi_initial_evidence`.
- **bolfi_posterior_samples** – Effective number of samples BOLFI will take from the posterior distribution. These are then used to fit a Gaussian to the posterior. Fit convergence scales with $1/\sqrt{n}$. Default: $I^2 + 3 * I$ with I as the number of estimated parameters. This is the number of the metaparameters of the underlying probability distribution times 2. The "times 2" considers the warmup samples.
- **ep_iterations** – The number of iterations of the Expectation Propagation algorithm, i.e., the number of passes over each feature. Default: 3.
- **ep_dampener** – The linear combination factor of the posterior calculated by BOLFI and the pseudo-prior. 0 means no dampening, i.e., the pseudo-prior gets replaced by the posterior. For values up to 1, that fraction of the pseudo-prior remains in each site update. Default: with a as the number of features and b as *ep_iterations*, `1 - a * (1 - ??√final_dampening)`.
- **final_dampening** – Alternative way to set *ep_dampener*. 0 means no dampening. For values up to 1, that fraction of the prior remains after the whole estimation. Default: if *ep_dampener* is not set, 0.5. Else, *ep_dampener* takes precedence.
- **ep_dampener_reduction_steps** – Number of iterations over which the *ep_dampener* gets reduced to 0. In each iteration, an equal fraction of it gets subtracted. Set to a negative number to disable the reduction. Default: -1.
- **gelman_rubin_threshold** – Optional threshold on top of the effective sample size. Values close to one indicate a converged estimate of the pseudo-posteriors. Never set to exactly one.
- **ess_ratio_sampling_from_zero** – Threshold in the ratio of effective sample size to samples in the pseudo-posterior estimation, at which the sampling defaults to starting at the center of the pseudo-prior. Set higher than *ess_ratio_resample* to disable this behaviour.
- **ess_ratio_resample** – Threshold in the ratio of effective sample size to samples in the pseudo-posterior estimation, at which before sampling the model gets resampled. Set higher than *ess_ratio_abort* to disable this behaviour.
- **ess_ratio_abort** – Threshold in the ratio of effective sample size to samples in the pseudo-posterior estimation, at which the sampling aborts and the pseudo-posterior update is skipped.
- **max_heuristic_steps** – The heuristics that are set by the *ess_ratio_x* arguments could effectively run forever. This parameter limits the amount of times these heuristics get employed in on EP iteration before it terminates.
- **posterior_sampling_increase** – The factor by which the ratio of the effective sample size to samples in the pseudo-posterior estimation is multiplied each loop (cumulatively). Never set to exactly one or lower, as it might result in an infinite loop.
- **model_resampling_increase** – The factor by which *bolfi_total_evidence* gets multiplied each time the model gets resampled.
- **independent_mcmc_chains** – The number of independent Markov-Chain Monte Carlo chains that are used for the estimation of the pseudo-posterior. Since we did not implement parallelization, more chains will not be faster, but more stable.

- **scramble_ep_feature_order** – True randomizes the order that the EP features are iterated over. Their order is still consistent across EP iterations. False uses the order that the *feature_extractors* define.
- **show_trials** – True plots the log of tried parameters live. Please note that each plot blocks the execution of the program, so do not use this when running the estimation in the background.
- **verbose** – True shows verbose error messages and logs of the estimation process. With False, you need to get the estimation results from *self.final_expectation* and *self.final_covariance*. Default: True.
- **seed** – Optional seed that is used in the RNG. If None is given, the results will be slightly different each time.

Returns

The BOLFI instance of the last EP iteration. As such, it contains the Posterior of the overall inference procedure.

simulator_index_by_feature

Mapping of index by all features to corresponding simulator.

sub_index_by_feature

Mapping of index by all features to that by one set of them.

visualize_parameter_distribution()

Plots the features and visualizes the correlation.

Please note that this function requires that the output of the individual simulators and the individual experimental data give an x- and y-axis when indexed with [0] and [1], respectively. Lists of lists in [0] and [1] with segmented data works as well. `EP_BOLFI.run` does not have these restrictions. Visualizes the comparison that EP_BOLFI was set up to infer model parameters with. May be used to check if everything works as intended. Additionally, the 95% confidence error bounds for the parameters are visualized to check for reasonable bounds. If called after `run`, the expected parameter set, the correlation and error bounds are from the finished estimation.

weights

Set the weights to unity if None are given.

ep_bolfi.optimization.EP_BOLFI.NDArrayEncoder

class ep_bolfi.optimization.EP_BOLFI.**NDArrayEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `JSONEncoder`

Use with the JSON library to store NumPy arrays.

__init__(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

Methods

__init__(*[, skipkeys, ensure_ascii, ...])	Constructor for JSONEncoder, with sensible defaults.
default(item)	Unpacks all NumPy arrays it finds into nested lists.
encode(o)	Return a JSON string representation of a Python data structure.
iterencode(o[, _one_shot])	Encode the given object and yield each string representation as available.

Attributes

item_separator
key_separator

default(*item*)

Unpacks all NumPy arrays it finds into nested lists.

Since this is called recursively, it only needs to check one level.

Parameters
item – Any object, which gets treated with the JSON library default if it is not a NumPy array. Else, it gets turned to a list.

Returns
A JSON encoded version of *item*.

encode(*o*)

Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*, *_one_shot=False*)

Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

ep_bolfi.optimization.EP_BOLFI.Optimizer_State

class ep_bolfi.optimization.EP_BOLFI.**Optimizer_State**(*input_dim*, *mcmc_chains*, *total_evidence*, *posterior_samples*, *gelman_rubin_threshold*, *ess_ratio_resample=5.0*, *ess_ratio_sampling_from_zero=-1.0*, *ess_ratio_abort=20.0*, *posterior_sampling_increase=1.2*, *model_resampling_increase=1.2*)

Bases: object

Handles the heuristics for the EP-BOLFI operation modes.

__init__(*input_dim*, *mcmc_chains*, *total_evidence*, *posterior_samples*, *gelman_rubin_threshold*, *ess_ratio_resample=5.0*, *ess_ratio_sampling_from_zero=-1.0*, *ess_ratio_abort=20.0*, *posterior_sampling_increase=1.2*, *model_resampling_increase=1.2*)

Methods

__init__(input_dim, mcmc_chains, ...[, ...])	
calculate_next_step(ess_ratio[, action])	

ep_bolfi.optimization.EP_BOLFI.Preprocessed_Simulator

class ep_bolfi.optimization.EP_BOLFI.**Preprocessed_Simulator**(*simulator*, *fixed_parameters*, *free_parameters_names*, *r*, *Q*, *experimental_data*, *feature_extractor*, *transform_parameters={}*, *fixed_parameter_order=None*)

Bases: object

Normalizes sampling to a standard normal distribution.

In order to help BOLFI to work efficiently with the least amount of setup required, this class mediates between the model parameters and a standard normal distribution for sampling. In a sense, the simulator output gets transformed into covariance eigenvectors.

__init__(*simulator*, *fixed_parameters*, *free_parameters_names*, *r*, *Q*, *experimental_data*, *feature_extractor*, *transform_parameters={}*, *fixed_parameter_order=None*)

Parameters

- **simulator** – The function that returns results given parameters.
- **fixed_parameters** – Dictionary of parameters that stay fixed and their values.
- **free_parameters_names** – List of the names of parameters which shall be inferred.
- **r** – *Q* times the mean of the distribution of free parameters.
- **Q** – Inverse covariance matrix of free parameters, the precision. It is used to transform the free parameters given to the 'simulator' into the ones used in the model. Most notably, these univariate standard normal distributions get transformed into a multivariate normal distribution corresponding to *Q* and *r*.
- **experimental_data** – The experimental data that the model will be fitted to. It has to have the same structure as the *simulator* output.
- **feature_extractor** – A function that takes the output of *simulator* or the *experimental_data* and returns a list of numbers.
- **transform_parameters** – Optional transformations between the parameter space that is used for searching for optimal parameters and the battery model parameters. *Q* and *r* define a normal distribution in that search space. The keys are the names of the free parameters. The values are 2-tuples. The first entry is a function taking the search space parameter and returning the model parameter. The second entry is the inverse function.
- **fixed_parameter_order** – Optional fixed parameter order. Prevents erroneous behaviour when the parameter dictionaries get reshuffled. Highly recommended.

Methods

__init__(simulator, fixed_parameters, ...[, ...])	
apply_transformation(trial_parameters)	Apply the transforms in *self.transform_parameters*.
elfi_simulator(*args, **kwargs)	A model simulator that can be used with ELFI.
search_to_transformed_trial(...)	Transforms search space parameters to model ones.
transformed_trial_to_search(...)	Transforms model space parameters to search ones.
undo_transformation(transformed_trial_parameters)	Undo the transforms in *self.transform_parameters*.

Attributes

log_of_tried_parameters	Stores all parameter combinations that have been tried.
experimental_features	Extract the features from the experimental data.
input_dim	Input dimension of the estimation task.
output_dim	Output dimension of the estimation task (number of features).
add_parameters	Create a function to combine the free and fixed parameters.
inv_variances	Compute the linear transformation of parameters for which the covariance of the underlying multivariate normal distribution is a diagonal matrix.
back_transform_matrix	Compute the linear transformation of parameters for which the covariance of the underlying multivariate normal distribution is a diagonal matrix.
variances	Variances of the model parameters.
transform_matrix	Inverse of *back_transform_matrix*.
transformed_means	`transform_matrix @ Q @ back_transform_matrix` is diagonal.
norm_factor	Now that the multivariate normal distribution is decomposed into various univariate ones, norm them to have equal variance 1.
un_norm_factor	Inverse of norm_factor.
normed_means	Expectation value of the normed univariate normal distributions.

add_parameters

Create a function to combine the free and fixed parameters.

apply_transformation (*trial_parameters*)

Apply the transforms in *self.transform_parameters*.

Parameters

trial_parameters – A dictionary. The keys are the *free_parameters_names* and the values are the actual model parameters.

Returns

The given dictionary with the vales transformed to the modified parameter space as specified in *self.transform_parameters*.

back_transform_matrix

Compute the linear transformation of parameters for which the covariance of the underlying multivariate normal distribution is a diagonal matrix. That is, compute the eigenvectors of Q. It is more stable since Q has growing eigenvectors in convergence.

elfi_simulator (**args*, ***kwargs*)

A model simulator that can be used with ELFI.

Parameters

- ***args** – The parameters as given by the prior nodes. Their order has to correspond to that of the parameter 'free_parameters' given to 'return_simulator'.
- ****kwargs** – Keyword parameters batch_size and random_state, but both are unused (they just get passed by BOLFI).

Returns
Simulated features for the given free parameters.

experimental_features

Extract the features from the experimental data.

input_dim

Input dimension of the estimation task.

inv_variances

Compute the linear transformation of parameters for which the covariance of the underlying multivariate normal distribution is a diagonal matrix. That is, compute the eigenvectors of Q. It is more stable since Q has growing eigenvectors in convergence.

log_of_tried_parameters

Stores all parameter combinations that have been tried.

norm_factor

Now that the multivariate normal distribution is decomposed into various univariate ones, norm them to have equal variance 1.

normed_means

Expectation value of the normed univariate normal distributions.

output_dim

Output dimension of the estimation task (number of features).

search_to_transformed_trial(*search_space_parameters*)

Transforms search space parameters to model ones.

Parameters
search_space_parameters – A list of lists which each contain a single search space parameter sample as it is returned by the sample functions of ELFI. In the case of only sample, a list also works.

Returns
A dictionary with its keys as the names of the parameters. Their order in the *search_space_parameters* is given by the order of *self.free_parameters_names*. The values yield the model parameters when passed through the functions in *self.transform_parameters*.

transform_matrix

Inverse of *back_transform_matrix*.

transformed_means

`transform_matrix @ Q @ back_transform_matrix` is diagonal. The correct transformation for vectors `v` is then `transform_matrix @ v`. The product below corresponds to `Q^-1 @ r`. It is just expressed in the eigenvector space of Q for efficiency.

transformed_trial_to_search(*model_space_parameters*)

Transforms model space parameters to search ones.

Parameters
model_space_parameters – A dictionary. The keys are the *self.free_parameters_names* and the values are the model parameters after applying the transformations given in *self.transform_parameters*.

Returns
A list (of lists) which each contain a single search space parameter sample as it is returned by the sample functions of ELFI. If the *model_space_parameters* dictionary values are numbers,

the returned value is a list. If they are lists, the returned value is a list of corresponding lists. In that case, each and every list must have the same length.

un_norm_factor

Inverse of norm_factor.

undo_transformation(*transformed_trial_parameters*)

Undo the transforms in *self.transform_parameters*.

Parameters

transformed_trial_parameters – A dictionary. The keys are the *free_parameters_names* and the values are the model parameters after they have been transformed as specified in *self.transform_parameters*.

Returns

The given dictionary with the values transformed back to the actual model parameter values.

variances

Variances of the model parameters.

2.1.4 ep_bolfi.utility

Modules

dataset_formatting	Defines datatypes for further processing.
fitting_functions	Various helper and fitting functions for processing measurement curves.
preprocessing	Contains frequently used workflows in dataset preprocessing.
visualization	Various helper and plotting functions for common data visualizations.

ep_bolfi.utility.dataset_formatting

Defines datatypes for further processing.

Functions

convert_none_notation_to_slicing(...)	Access a slice of an HDF5 object by transferable notation.
get_hdf5_dataset_by_path(h5py_object, path)	Follow the structure of a HDF object to get a certain part.
print_hdf5_structure(h5py_object[, depth, ...])	Simple HDF5 structure viewer.
read_csv_from_measurement_system(path, ...)	Read the measurements as returned by common instruments.
read_hdf5_table(path, data_location, headers)	Read the measurements as stored in a HDF5 file.
read_parquet_table(file_name, datatype)	Read an Apache Parquet serialization of a `Measurement` object.
store_parquet_table(measurement, file_prefix)	Store an Apache Parquet serialization of a `Measurement` object.

ep_bolfi.utility.dataset_formatting.convert_none_notation_to_slicing

ep_bolfi.utility.dataset_formatting.**convert_none_notation_to_slicing**(*h5py_object*, *index*)

Access a slice of an HDF5 object by transferable notation.

Parameters

- **h5py_object** – The HDF5 object wrapped by H5Py, for example `h5py.File(filename, 'r')`.
- **index** – A 2-tuple or a 2-list. (None, x) denotes slicing [:, x] and (x, None) denotes slicing [x, :].

Returns

The None-notation sliced h5py object.

ep_bolfi.utility.dataset_formatting.get_hdf5_dataset_by_path

ep_bolfi.utility.dataset_formatting.**get_hdf5_dataset_by_path**(*h5py_object*, *path*)

Follow the structure of a HDF object to get a certain part.

Parameters

- **h5py_object** – The HDF5 object wrapped by H5Py, for example `h5py.File(filename, 'r')`.
- **path** – A list. Each entry goes one level deeper into the HDF structure. Each entry can either be the index to go into next itself, or a 2-tuple or a 2-list. In the latter case, (None, x) denotes slicing [:, x] and (x, None) denotes slicing [x, :].

Returns

Returns the HDF5 object found at the end of *path*.

ep_bolfi.utility.dataset_formatting.print_hdf5_structure

ep_bolfi.utility.dataset_formatting.**print_hdf5_structure**(*h5py_object*, *depth=1*, *table_limit=4*, *verbose_limit=64*)

Simple HDF5 structure viewer.

Parameters

- **h5py_object** – The HDF5 object wrapped by H5Py, for example `h5py.File(filename, 'r')`.
- **depth** – For pretty printing the recursive depth. Do not change.
- **table_limit** – h5py.Dataset object tables will be truncated prior to printing up to this number in the higher dimension.
- **verbose_limit** – h5py.Dataset objects will be truncated to at most this number in any dimension.

ep_bolfi.utility.dataset_formatting.read_csv_from_measurement_system

ep_bolfi.utility.dataset_formatting.**read_csv_from_measurement_system**(*path*, *encoding*, *number_of_comment_lines*, *headers*, *delimiter='\t'*, *decimal='.'*, *datatype='cycling'*, *segment_column=-1*, *segments_to_process=None*, *current_sign_correction={}*, *correction_column=-1*, *flip_voltage_sign=False*, *flip_imaginary_impedance_sign=False*, *max_number_of_lines=-1*)

Read the measurements as returned by common instruments.

Example: cycling measurements from Basytec devices. Their format resembles a csv file with one title and one header comment line. So the first line will be ignored and the second used for headers.

Parameters

- **path** – The full or relative path to the measurement file.
- **encoding** – The encoding of that file, e.g. “iso-8859-1”.
- **number_of_comment_lines** – The number of lines that have to be skipped over in order to arrive at the first dataset line.
- **headers** – A dictionary. Its keys are the indices of the columns which are to be read in. The corresponding values are there to tell this function which kind of data is in which column. The following format has to be used: “<name> [<unit>]” where “name” is “U” (voltage), “I” (current), or “t” (time) and “unit” is “V”, “A”, “h”, “m” ,or “s” with the optional prefixes “k”, “m”, “µ”, or “n”. This converts the data to prefix-less SI units. Additional columns may be read in with keys not in this format. The columns for segments and sign correction are only given by *segment_column* and *correction_column*.
- **delimiter** – The delimiter string between datapoints. The default is “ “.
- **decimal** – The string used for the decimal point. Default: “.”.
- **datatype** – Default is “cycling”, where cycling information is assumed in the file. “static” will trigger the additional extraction of exponential decays that are relevant to e.g. GITT.

"impedance" will treat the file as an impedance measurement with frequencies and impedances instead of time and voltage.

- **segment_column** – The index of the column that stores the index of the current segment. If it changes from one data point to the next, that is used as the dividing line between two segments. Default is -1, which returns the dataset in one segment.
- **segments_to_process** – A list of indices which give the segments that shall be processed. Default is None, i.e., the whole file gets processed.
- **current_sign_correction** – A dictionary. Its keys are the strings used in the file to indicate a state. The column from which this state is retrieved is given by *correction_column*. The dictionaries' values are used to correct/normalize the current value in the file. For example, if discharge currents have the same positive sign as charge currents in the file, use -1 to correct that, or if the values are to be scaled by weight, use the scaling factor. The default is the empty dictionary.
- **correction_column** – See #current_sign_correction. Default: -1.
- **max_number_of_lines** – The maximum number of dataset lines that are to be read in. Default: -1 (no limit).
- **flip_voltage_sign** – Defaults to False, where measured voltage remains unaltered. Change to True if the voltage shall be multiplied by -1. Also applies to impedances; real and imaginary parts.
- **flip_imaginary_impedance_sign** – Defaults to False, where measured impedance remains unaltered. Change to True if the imaginary part of the impedance shall be multiplied by -1. Cancels out with *flip_voltage_sign*.

Returns

The measurement, packaged in a Measurement subclass. It depends on "datatype" which one it is:

- "cycling": `Cycling_Information`
- "static": `Static_Information`
- "impedance": `Impedance_Measurement`

ep_bolfi.utility.dataset_formatting.read_hdf5_table

ep_bolfi.utility.dataset_formatting.**read_hdf5_table**(*path*, *data_location*, *headers*, *datatype='cycling'*, *segment_location=None*, *segments_to_process=None*, *current_sign_correction={}*, *correction_location=None*, *flip_voltage_sign=False*, *flip_imaginary_impedance_sign=False*)

Read the measurements as stored in a HDF5 file.

Parameters

- **path** – The full or relative path to the measurement file.
- **data_location** – A list. Gives the location in the HDF5 file where the data table is stored. Set to None if everything is stored at the top level. Each entry goes one level deeper into the HDF structure. Each entry can either be the index to go into next itself, or a 2-tuple or a 2-list. In the latter case, (None, x) denotes slicing [:, x] and (x, None) denotes slicing [x, :].

- **headers** – A dictionary. Its keys are 2-tuples, slicing the data which are to be read in. Use the format "(x, None)" or "(None, x)" to slice the dimension with "None". The corresponding values are there to tell this function which kind of data is in which column. If necessary, the keys may also be tuples like *data_location*. The following format has to be used: "<name> [<unit>]" where "name" is "U" (voltage), "I" (current), or "t" (time) and "unit" is "V", "A", "h", "m", or "s" with the optional prefixes "k", "m", "µ", or "n". This converts the data to prefix-less SI units. Additional columns may be read in with keys not in this format. The columns for segments and sign correction are only given by *segment_column* and *correction_column*.
- **datatype** – Default is "cycling", where cycling information is assumed in the file. "static" will trigger the additional extraction of exponential decays that are relevant to e.g. GITT. "impedance" will treat the file as an impedance measurement with frequencies and impedances instead of time and voltage.
- **segments_to_process** – A list of indices which give the segments that shall be processed. Default is None, i.e., the whole file gets processed.
- **current_sign_correction** – A dictionary. Its keys are the strings used in the file to indicate a state. The column from which this state is retrieved is given by *correction_column*. The dictionaries' values are used to correct/normalize the current value in the file. For example, if discharge currents have the same positive sign as charge currents in the file, use -1 to correct that, or if the values are to be scaled by weight, use the scaling factor. The default is the empty dictionary.
- **correction_location** – A list, with the same format as *data_location*. For its use, see *current_sign_correction*. Default: None.
- **flip_voltage_sign** – Defaults to False, where measured voltage remains unaltered. Change to True if the voltage shall be multiplied by -1. Also applies to impedances; real and imaginary parts.
- **flip_imaginary_impedance_sign** – Defaults to False, where measured impedance remains unaltered. Change to True if the imaginary part of the impedance shall be multiplied by -1. Cancels out with *flip_voltage_sign*.

Segment_location

A list, with the same format as *data_location*. It points to the part of the data that stores the index of the current segment. If it changes from one data point to the next, that is used as the dividing line between two segments. Default is None, which returns the dataset in one segment.

Returns

The measurement, packaged in a Measurement subclass. It depends on "datatype" which one it is:

- "cycling": `Cycling_Information`
- "static": `Static_Information`
- "impedance": `Impedance_Measurement`

ep_bolfi.utility.dataset_formatting.read_parquet_table

ep_bolfi.utility.dataset_formatting.**read_parquet_table**(*file_name*, *datatype*)

Read an Apache Parquet serialization of a `Measurement` object. For storing such a serialization, refer to `store_parquet_table`.

Parameters

- **file_name** – The full name of the file to read from.
- **datatype** –

  One of the following, denoting what `Measurement` was stored:

  – 'cycling': `Cycling_Information`

  – 'static': `Static_Information`

  – 'impedance': `Impedance_Information`

Returns

The `Measurement` object of choice.

ep_bolfi.utility.dataset_formatting.store_parquet_table

ep_bolfi.utility.dataset_formatting.**store_parquet_table**(*measurement*, *file_prefix*, *compression_level=22*)

Store an Apache Parquet serialization of a `Measurement` object. For reading such a serialization, refer to `read_parquet_table`.

Parameters

- **measurement** – The `Measurement` object to serialize.
- **file_prefix** – The filename to write into. A '.parquet' ending is appended.
- **compression_level** – Compression level of the compression algorithm Zstandard. -7 gives the largest files with highest compression speed. 22 gives the smallest files with slowest compression speed. Note that decompression has the same (fast) speed at any level.

Classes

Cycling_Information(timepoints, currents, ...)	Contains basic cycling informations.
Impedance_Measurement(frequencies, ...[, ...])	Contains basic impedance data.
Measurement()	Defines common methods for measurement objects.
Static_Information(timepoints, currents, ...)	Contains additional informations, e.g. for GITT.

ep_bolfi.utility.dataset_formatting.Cycling_Information

class ep_bolfi.utility.dataset_formatting.**Cycling_Information**(*timepoints*, *currents*, *voltages*, *other_columns={}*, *indices=None*)

Bases: *Measurement*

Contains basic cycling informations. Each member variable is a list and has the same length as the other ones.

__init__(*timepoints*, *currents*, *voltages*, *other_columns={}*, *indices=None*)

Methods

__init__(timepoints, currents, voltages[, ...])	
example_table_row()	Makes singular *self.indices* entries into lists and returns an exempular line for table formatting.
extend(other)	Extends *self* with the *other* `Measurement` object.
from_json(json_string)	Reads this object from JSON that was prepared with `to_json`.
segment_tables([start, stop, step])	Iterator that gives data segments in Apache Parquet syntax.
subarray(array)	Returns a `Measurement` object containing *array* segments.
subslice(start, stop[, step])	Returns a `Measurement` object containing the requested slice.
table_descriptors()	Gives the headings for table formatting.
table_mapping()	Gives the mapping from attributes to headings.
to_json()	Losslessly transforms this object into JSON.

Attributes

timepoints	The times at which measurements were taken.
currents	The measured current at those times.
voltages	The measured voltage at those times.
other_columns	The contents of any other columns.
indices	The indices of the individual segments.

currents

The measured current at those times. A list which usually contains lists for segments with variable current and floats for segments with constant current.

example_table_row()

Makes singular *self.indices* entries into lists and returns an exempular line for table formatting.

Returns

A list with the entries for each column.

extend(*other*)

Extends *self* with the *other* `Measurement` object.

Parameters
other – Another `Measurement object`.

classmethod from_json(*json_string*)

Reads this object from JSON that was prepared with `to_json`.

Parameters
json_string – The JSON, fed in as a string.

indices

The indices of the individual segments. Defaults to a simple numbering of the segments present. May be used for plotting purposes, e.g., for colourcoding the segments by cycle.

other_columns

The contents of any other columns. A dictionary ("columns") which values are lists which contain lists for segments. The keys should match user input for the columns.

segment_tables(*start=0*, *stop=None*, *step=1*)

Iterator that gives data segments in Apache Parquet syntax.

Parameters

- **start** – First data segment.
- **stop** – Last data segment.
- **step** – Step size in each iteration.

subarray(*array*)

Returns a `Measurement` object containing *array* segments.

Parameters
array – A list of integers denoting the segments to collect. Does not correspond to *self.indices*.

Returns
A `Measurement[array]` object.

subslice(*start*, *stop*, *step=1*)

Returns a `Measurement` object containing the requested slice.

Parameters

- **start** – First data segment.
- **stop** – Last data segment.
- **step** – Step size in each iteration.

Returns
A `Measurement[start:stop:step]` object.

table_descriptors()

Gives the headings for table formatting.

Returns
Table headings in Apache Parquet syntax.

classmethod table_mapping()

Gives the mapping from attributes to headings.

Returns
A dictionary: keys are attribute names, values are headings.

timepoints

The times at which measurements were taken. Usually a list of lists where each list corresponds to a segment of the measurement.

to_json()

Losslessly transforms this object into JSON.

Returns
The JSON representation, given as a string.

voltages

The measured voltage at those times. A list which usually contains lists for segments with variable voltage and floats for segments with constant voltage.

ep_bolfi.utility.dataset_formatting.Impedance_Measurement

class ep_bolfi.utility.dataset_formatting.**Impedance_Measurement**(*frequencies*, *real_impedances*, *imaginary_impedances*, *phases*, *other_columns={}*, *indices=None*)

Bases: *Measurement*

Contains basic impedance data. Each member variable is a list and has the same length as the other ones.

__init__(*frequencies*, *real_impedances*, *imaginary_impedances*, *phases*, *other_columns={}*, *indices=None*)

Methods

__init__(frequencies, real_impedances, ...)	
example_table_row()	Makes singular *self.indices* entries into lists and returns an exempular line for table formatting.
extend(other)	Extends *self* with the *other* Measurement object.
from_json(json_string)	Reads this object from JSON that was prepared with to_json.
segment_tables([start, stop, step])	Iterator that gives data segments in Apache Parquet syntax.
subarray(array)	Returns a Measurement object containing *array* segments.
subslice(start, stop[, step])	Returns a Measurement object containing the requested slice.
table_descriptors()	Gives the headings for table formatting.
table_mapping()	Gives the mapping from attributes to headings.
to_json()	Losslessly transforms this object into JSON.

Attributes

complex_impedances	
frequencies	The frequencies at which impedances were measured.
real_impedances	The real part of the impedances measured at those frequencies.
imaginary_impedances	The imaginary part of the impedances measured at those frequencies.
phases	The phases of the impedance measured at those frequencies.
other_columns	The contents of any other columns.
indices	The indices of the individual segments.

example_table_row()

Makes singular *self.indices* entries into lists and returns an exempular line for table formatting.

Returns
A list with the entries for each column.

extend(*other*)

Extends *self* with the *other* `Measurement` object.

Parameters
other – Another `Measurement object`.

frequencies

The frequencies at which impedances were measured. Usually a list of lists where each list corresponds to a different equilibrium.

classmethod from_json(*json_string*)

Reads this object from JSON that was prepared with `to_json`.

Parameters
json_string – The JSON, fed in as a string.

imaginary_impedances

The imaginary part of the impedances measured at those frequencies.

indices

The indices of the individual segments. Defaults to a simple numbering of the segments present. May be used for plotting purposes, e.g., for colourcoding the segments by cycle.

other_columns

The contents of any other columns. A dictionary ("columns") which values are lists which contain lists for segments. The keys should match user input for the columns.

phases

The phases of the impedance measured at those frequencies.

real_impedances

The real part of the impedances measured at those frequencies.

segment_tables(*start=0*, *stop=None*, *step=1*)

Iterator that gives data segments in Apache Parquet syntax.

Parameters

- **start** – First data segment.
- **stop** – Last data segment.
- **step** – Step size in each iteration.

subarray(*array*)

Returns a `Measurement` object containing *array* segments.

Parameters
array – A list of integers denoting the segments to collect. Does not correspond to *self.indices*.

Returns
A `Measurement[array]` object.

subslice(*start*, *stop*, *step=1*)

Returns a `Measurement` object containing the requested slice.

Parameters

- **start** – First data segment.
- **stop** – Last data segment.
- **step** – Step size in each iteration.

Returns
A `Measurement[start:stop:step]` object.

table_descriptors()

Gives the headings for table formatting.

Returns
Table headings in Apache Parquet syntax.

classmethod table_mapping()

Gives the mapping from attributes to headings.

Returns
A dictionary: keys are attribute names, values are headings.

to_json()

Losslessly transforms this object into JSON.

Returns
The JSON representation, given as a string.

ep_bolfi.utility.dataset_formatting.Measurement

class ep_bolfi.utility.dataset_formatting.**Measurement**

Bases: `object`

Defines common methods for measurement objects.

__init__()

Methods

__init__()	
example_table_row()	Makes singular *self.indices* entries into lists and returns an exempular line for table formatting.
extend(other)	Extends *self* with the *other* `Measurement` object.
from_json(json_string)	Reads this object from JSON that was prepared with `to_json`.
segment_tables([start, stop, step])	Iterator that gives data segments in Apache Parquet syntax.
subarray(array)	Returns a `Measurement` object containing *array* segments.
subslice(start, stop[, step])	Returns a `Measurement` object containing the requested slice.
table_descriptors()	Gives the headings for table formatting.
table_mapping()	Gives the mapping from attributes to headings.
to_json()	Losslessly transforms this object into JSON.

example_table_row()

Makes singular *self.indices* entries into lists and returns an exempular line for table formatting.

Returns
A list with the entries for each column.

extend(*other*)

Extends *self* with the *other* `Measurement` object.

Parameters
other – Another `Measurement object`.

classmethod from_json(*json_string*)

Reads this object from JSON that was prepared with `to_json`.

Parameters
json_string – The JSON, fed in as a string.

segment_tables(*start=0*, *stop=None*, *step=1*)

Iterator that gives data segments in Apache Parquet syntax.

Parameters

- **start** – First data segment.
- **stop** – Last data segment.
- **step** – Step size in each iteration.

subarray(*array*)

Returns a `Measurement` object containing *array* segments.

Parameters
array – A list of integers denoting the segments to collect. Does not correspond to *self.indices*.

Returns
A `Measurement[array]` object.

subslice(*start*, *stop*, *step=1*)

Returns a `Measurement` object containing the requested slice.

Parameters

- **start** – First data segment.
- **stop** – Last data segment.
- **step** – Step size in each iteration.

Returns

A `Measurement[start:stop:step]` object.

table_descriptors()

Gives the headings for table formatting.

Returns

Table headings in Apache Parquet syntax.

classmethod table_mapping()

Gives the mapping from attributes to headings.

Returns

A dictionary: keys are attribute names, values are headings.

to_json()

Losslessly transforms this object into JSON.

Returns

The JSON representation, given as a string.

ep_bolfi.utility.dataset_formatting.Static_Information

class ep_bolfi.utility.dataset_formatting.**Static_Information**(*timepoints*, *currents*, *voltages*, *other_columns={}*, *indices=None*)

Bases: *Cycling_Information*

Contains additional informations, e.g. for GITT. Each member variable is a list and has the same length as the other ones.

__init__(*timepoints*, *currents*, *voltages*, *other_columns={}*, *indices=None*)

Methods

__init__(timepoints, currents, voltages[, ...])	
example_table_row()	Makes singular *self.indices* entries into lists and returns an exempular line for table formatting.
extend(other)	Extends *self* with the *other* `Measurement` object.
from_json(json_string)	Reads this object from JSON that was prepared with `to_json`.
segment_tables([start, stop, step])	Iterator that gives data segments in Apache Parquet syntax.
subarray(array)	Returns a `Measurement` object containing *array* segments.
subslice(start, stop[, step])	Returns a `Measurement` object containing the requested slice.
table_descriptors()	Gives the headings for table formatting.
table_mapping()	Gives the mapping from attributes to headings.
to_json()	Losslessly transforms this object into JSON.

Attributes

asymptotic_voltages	The voltages that the voltage curve seems to converge to in a segment.
ir_steps	The instantaneous IR drops before each segment.
exp_I_decays	Same as exp_U_decays for current decays (PITT).
exp_U_decays	The fit parameters of the exponential voltage decays in each segment. Each set of fit parameters is a 3-tuple (a,b,c) where the fit function has the following form: a + b * exp(-c * (t - t_end_of_segment)). Failed or missing fits are best indicated by (NaN, NaN, NaN).
timepoints	The times at which measurements were taken.
currents	The measured current at those times.
voltages	The measured voltage at those times.
other_columns	The contents of any other columns.
indices	The indices of the individual segments.

asymptotic_voltages

The voltages that the voltage curve seems to converge to in a segment. Only makes sense for those segments that are rest periods or when the OCV was subtracted.

currents

The measured current at those times. A list which usually contains lists for segments with variable current and floats for segments with constant current.

example_table_row()

Makes singular *self.indices* entries into lists and returns an exempular line for table formatting.

Returns

A list with the entries for each column.

exp_I_decays

Same as exp_U_decays for current decays (PITT).

exp_U_decays

The fit parameters of the exponential voltage decays in each segment. Each set of fit parameters is a 3-tuple (a,b,c) where the fit function has the following form:

a + b * exp(-c * (t - t_end_of_segment)).

Failed or missing fits are best indicated by (NaN, NaN, NaN).

extend(*other*)

Extends *self* with the *other* `Measurement` object.

Parameters
other – Another `Measurement object`.

classmethod from_json(*json_string*)

Reads this object from JSON that was prepared with `to_json`.

Parameters
json_string – The JSON, fed in as a string.

indices

The indices of the individual segments. Defaults to a simple numbering of the segments present. May be used for plotting purposes, e.g., for colourcoding the segments by cycle.

ir_steps

The instantaneous IR drops before each segment. Positive values are voltage rises and negative values voltage drops.

other_columns

The contents of any other columns. A dictionary ("columns") which values are lists which contain lists for segments. The keys should match user input for the columns.

segment_tables(*start=0*, *stop=None*, *step=1*)

Iterator that gives data segments in Apache Parquet syntax.

Parameters

- **start** – First data segment.
- **stop** – Last data segment.
- **step** – Step size in each iteration.

subarray(*array*)

Returns a `Measurement` object containing *array* segments.

Parameters
array – A list of integers denoting the segments to collect. Does not correspond to *self.indices*.

Returns
A `Measurement[array]` object.

subslice(*start*, *stop*, *step=1*)

Returns a `Measurement` object containing the requested slice.

Parameters

- **start** – First data segment.
- **stop** – Last data segment.
- **step** – Step size in each iteration.

Returns
A `Measurement[start:stop:step]` object.

table_descriptors()

Gives the headings for table formatting.

Returns
Table headings in Apache Parquet syntax.

classmethod table_mapping()

Gives the mapping from attributes to headings.

Returns
A dictionary: keys are attribute names, values are headings.

timepoints

The times at which measurements were taken. Usually a list of lists where each list corresponds to a segment of the measurement.

to_json()

Losslessly transforms this object into JSON.

Returns
The JSON representation, given as a string.

voltages

The measured voltage at those times. A list which usually contains lists for segments with variable voltage and floats for segments with constant voltage.

ep_bolfi.utility.fitting_functions

Various helper and fitting functions for processing measurement curves.

Functions

OCV_fit_function(E_OCV, *args[, z, T, ...])	The OCV model from "A parametric OCV model".
a_fit(γUeminus1)	Calculates the conversion from "A parametric OCV model" (Birkl).
d2_dE2_OCV_fit_function(E_OCV, *args[, z, ...])	The 2nd derivative of `fitting_functions.OCV_fit_function`.
d_dE_OCV_fit_function(E_OCV, *args[, z, T, ...])	The derivative of `fitting_functions.OCV_fit_function`.
find_occurrences(sequence, value)	Gives indices in sequence where it is closest to value.
fit_OCV(SOC, OCV[, N, SOC_range_bounds, ...])	Fits data to `fitting_functions.OCV_fit_function`.
fit_drt(frequencies, impedances[, lambda_value])	A Distribution of Relaxation Times approximation via `pyimpspec`.
fit_exponential_decay(timepoints, voltages)	See `fit_exponential_decay_with_warnings` for details.
fit_exponential_decay_with_warnings(...[, ...])	Extracts a set amount of exponential decay curves.
fit_pwrlaw(timepoints, quantity[, threshold])	Extracts a powerlaw from the data..
fit_pwrlawCL(timepoints, quantity[, threshold])	Extracts a powerlaw from the data.
fit_sqrt(timepoints, voltages[, threshold])	See `fit_sqrt_with_warnings` for details.
fit_sqrt_with_warnings(timepoints, voltages)	Extracts a square root at the beginning of the data.
inverse_OCV_fit_function(SOC, *args[, z, T, ...])	The inverse of `fitting_functions.OCV_fit_function`.
inverse_d2_dSOC2_OCV_fit_function(SOC, *args)	The 2nd derivative of the inverse of `OCV_fit_function`.
inverse_d_dSOC_OCV_fit_function(SOC, *args)	
laplace_transform(x, y, s)	Performs a basic Laplace transformation.
smooth_fit(x, y[, order, splits, w, s, ...])	Calculates a smoothed spline with derivatives.
verbose_spline_parameterization(coeffs, ...)	Gives the monomic representation of a B-spline.

ep_bolfi.utility.fitting_functions.OCV_fit_function

ep_bolfi.utility.fitting_functions.**OCV_fit_function**(*E_OCV*, **args*, *z=1.0*, *T=298.15*, *individual=False*, *fit_SOC_range=False*, *rescale=False*)

The OCV model from "A parametric OCV model".

Reference

C. R. Birkl, E. McTurk, M. R. Roberts, P. G. Bruce and D. A. Howey. "A Parametric Open Circuit Voltage Model for Lithium Ion Batteries". Journal of The Electrochemical Society, 162(12):A2271-A2280, 2015

param E_OCV
The voltages for which the SOCs shall be evaluated.

param *args
A list which length is dividable by three. These are the parameters E_0, a and Δx from the paper referenced above in the order (E_0_0, a_0, Δx_0, E_0_1, a_1, Δx_1…). If *fit_SOC_range* is True,

two additional arguments are at the front (see there). The last Δx_j may be omitted to force $\sum_j$ Δx_j = 1.

param z
The charge number of the electrode interface reaction.

param T
The temperature of the electrode.

param individual
If True, the model function summands are not summed up.

param fit_SOC_range
If True, this function takes two additional arguments at the start of *args* that may be used to adjust the data SOC range.

param rescale
If True, the expected *args* now contain the slopes of the summands at their respective origins instead of a. Formula: `a / slope = -4 * (k_B / e) * T / (Δx * z)`.

returns
The evaluation of the model function. This is the referenced fit function on a linearly transformed SOC range if *fit_SOC_range* is True. If *individual* is True, the individual summands in the model function get returned as a list.

ep_bolfi.utility.fitting_functions.a_fit

ep_bolfi.utility.fitting_functions.**a_fit**(*γUeminus1*)

Calculates the conversion from "A parametric OCV model" (Birkl).

Parameters
yUeminus1 – γ * U_i / e (see "A parametric OCV model").

Returns
The approximation factor a_i from "A parametric OCV model".

ep_bolfi.utility.fitting_functions.d2_dE2_OCV_fit_function

ep_bolfi.utility.fitting_functions.**d2_dE2_OCV_fit_function**(*E_OCV*, **args*, *z=1.0*, *T=298.15*, *individual=False*, *fit_SOC_range=False*, *rescale=False*)

The 2nd derivative of `fitting_functions.OCV_fit_function`.

Parameters

- **E_OCV** – The voltages for which the 2nd derivative shall be evaluated.
- ***args** – A list which length is dividable by three. These are the parameters E_0, a and Δx from "A parametric OCV model" in the order (E_0_0, a_0, Δx_0, E_0_1, a_1, Δx_1…). The last Δx_j may be omitted to force $\sum_j$ Δx_j = 1.
- **z** – The charge number of the electrode interface reaction.
- **T** – The temperature of the electrode.
- **individual** – If True, the model function summands are not summed up.

- **fit_SOC_range** – If True, this function takes two additional arguments at the start of *args* that may be used to adjust the data SOC range.
- **rescale** – If True, the expected *args* now contain the slopes of the summands at their respective origins instead of a. Formula: `a / slope = -4 * (k_B / e) * T / (Δx * z)`.

Returns
The evaluation of ∂^2OCV_fit_function(OCV) / ∂OCV^2.

ep_bolfi.utility.fitting_functions.d_dE_OCV_fit_function

ep_bolfi.utility.fitting_functions.**d_dE_OCV_fit_function**(*E_OCV*, **args*, *z=1.0*, *T=298.15*, *individual=False*, *fit_SOC_range=False*, *rescale=False*)

The derivative of `fitting_functions.OCV_fit_function`.

Parameters

- **E_OCV** – The voltages for which the derivative shall be evaluated.
- ***args** – A list which length is dividable by three. These are the parameters E_0, a and Δx from "A parametric OCV model" in the order (E_0_0, a_0, Δx_0, E_0_1, a_1, Δx_1…). The last Δx_j may be omitted to force $\sum_j \Delta x\_j = 1$.
- **z** – The charge number of the electrode interface reaction.
- **T** – The temperature of the electrode.
- **individual** – If True, the model function summands are not summed up.
- **fit_SOC_range** – If True, this function takes two additional arguments at the start of "args" that may be used to adjust the data SOC range.
- **rescale** – If True, the expected "args" now contain the slopes of the summands at their respective origins instead of a. Formula: `a / slope = -4 * (k_B / e) * T / (Δx * z)`.

Returns
The evaluation of ∂OCV_fit_function(OCV) / ∂OCV.

ep_bolfi.utility.fitting_functions.find_occurrences

ep_bolfi.utility.fitting_functions.**find_occurrences**(*sequence*, *value*)

Gives indices in sequence where it is closest to value.

Parameters

- **sequence** – A list that represents a differentiable function.
- **value** – The value that is searched for in *sequence*. Also, crossings of consecutive values in *sequence* with *value* are searched for.

Returns
A list of indices in *sequence* in ascending order where *value* or a close match for *value* was found.

ep_bolfi.utility.fitting_functions.fit_OCV

ep_bolfi.utility.fitting_functions.**fit_OCV**(*SOC*, *OCV*, *N=4*, *SOC_range_bounds=(0.2, 0.8)*, *SOC_range_limits=(0.0, 1.0)*, *z=1.0*, *T=298.15*, *inverted=True*, *fit_SOC_range=True*, *distance_order=2*, *weights=None*, *initial_parameters=None*, *minimize_options=None*)

Fits data to `fitting_functions.OCV_fit_function`.

In addition to the fit itself, a model-based correction to the provided SOC-OCV-data is made. If *SOC* lives in a (0,1)-range, the correction is given as its transformation to the (SOC_start, SOC_end)-range given as the first two returned numbers.

Parameters

- **SOC** – The SOCs at which measurements were made.
- **OCV** – The corresponding open-circuit voltages.
- **N** – The number of phases of the OCV model.
- **SOC_range_bounds** – Optional hard upper and lower bounds for the SOC correction from the left and the right side, respectively, as a 2-tuple. Use it as a limiting guess for the actual SOC range represented in the measurement. Has to be inside (0.0, 1.0). Set to (0.0, 1.0) to effectively disable SOC range estimation.
- **SOC_range_limits** – Optional hard lower and upper bounds for the SOC correction from the left and the right side, respectively, as a 2-tuple. Use it if you know that your OCV data is incomplete and by how much. Has to be inside (0.0, 1.0). Set to (0.0, 1.0) to allow the SOC range estimation to assign datapoints to the asymptotes.
- **z** – The charge number of the electrode interface reaction.
- **T** – The temperature of the electrode.
- **inverted** – If True (default), the widely adopted SOC convention is assumed. If False, the formulation of "A parametric OCV model" is used.
- **fit_SOC_range** – If True (default), a model-based correction to the provided SOC-OCV-data is made.
- **distance_order** – The order of the norm of the vector of the distances between OCV data and OCV model. Default is 2, i.e., the Euclidean norm. 1 sets it to absolute distance, and `float('inf')` sets it to maximum distance. Note that 1 will lead to worse performance.
- **weights** – Optional weights to apply to the vector of the distances between OCV data and OCV model. Defaults to equal weights.
- **initial_parameters** – Optional initial guess for the model parameters. If left as-is, this will be automatically gleaned from the data. Use only if you have another fit to data of the same electrode material.
- **minimize_options** – Dictionary that gets passed to scipy.optimize.minimize with the method `trust-constr`. See scipy.optimize.show_options with the arguments 'minimize' and 'trust-constr' for details.

Returns

The fitted parameters of `fitting_functions.OCV_fit_function` plus the fitted SOC range prepended.

ep_bolfi.utility.fitting_functions.fit_drt

ep_bolfi.utility.fitting_functions.**fit_drt**(*frequencies*, *impedances*, *lambda_value=-2.0*)

A Distribution of Relaxation Times approximation via pyimpspec.

Parameters

- **frequencies** – Array of the measured frequencies.
- **impedances** – Array of the measured complex impedances.
- **lambda_value** – Takes the place of the R^2 value as a tuning parameter. Consult the pyimpspec documentation for the precise usage. Default is -2, which "uses the L-curve approach to estimate λ". -1 uses a different heuristic, and values > 0 set λ directly.

Returns

A 3-tuple of characteristic DRT time constants, their corresponding resistances, and the whole TRNNLSResult object returned by pyimpspec. Note that the .pseudo_chisqr attribute is not useful for the same reasons that went into the R^2 calculations in the other fit functions: a "pseudo-R^2" does not use the inverse function to obtain a sensible R^2.

ep_bolfi.utility.fitting_functions.fit_exponential_decay

ep_bolfi.utility.fitting_functions.**fit_exponential_decay**(*timepoints*, *voltages*, *recursive_depth=1*, *threshold=0.95*)

See fit_exponential_decay_with_warnings for details. This method does the same, but suppresses inconsequential NumPy warnings.

ep_bolfi.utility.fitting_functions.fit_exponential_decay_with_warnings

ep_bolfi.utility.fitting_functions.**fit_exponential_decay_with_warnings**(*timepoints*, *voltages*, *recursive_depth=1*, *threshold=0.95*)

Extracts a set amount of exponential decay curves.

Parameters

- **timepoints** – The timepoints of the measurements.
- **voltages** – The corresponding voltages.
- **recursive_depth:** – The default 1 fits one exponential curve to the data. For higher values that fit is repeated with the data minus the preceding fit(s) for this amount of times minus one.
- **threshold** – The lower threshold value for the R^2 coefficient of determination. If *threshold* is smaller than 1, the subset of the exponential decay data is searched that just fulfills it. Defaults to 0.95. Values above 1 are set to 1.

Returns

A list of length *recursive_depth* where each element is a 3-tuple with the timepoints, the fitted

voltage evaluations and a 3-tuple of the parameters of the following decay function: `t, (U_0, ΔU, τ⁻¹): U_0 + ΔU * np.exp(-τ⁻¹ * (t - timepoints[0]))`.

ep_bolfi.utility.fitting_functions.fit_pwrlaw

ep_bolfi.utility.fitting_functions.**fit_pwrlaw**(*timepoints*, *quantity*, *threshold=0.95*)

Extracts a powerlaw from the data..

Parameters

- **timepoints** – The timepoints of the measurements.
- **quantity** – The corresponding quantity, to which a powerlaw-behaviour should be fitted.
- **threshold** – The lower threshold value for the R^2 coefficient of determination. If *threshold* is larger than the R^2 coefficient, a warning is issued and the parameters may be NaN or Inf. Defaults to 0.95. Values above 1 are set to 0.98.

Returns

A 3-tuple with the timepoints, the fitted quantity evaluations and a 4-tuple of the parameters of the following powerlaw: `t, (q0, dq_dt, q_trans, n): q0 + dq_dt * (t - q_trans)**n`.

ep_bolfi.utility.fitting_functions.fit_pwrlawCL

ep_bolfi.utility.fitting_functions.**fit_pwrlawCL**(*timepoints*, *quantity*, *threshold=0.95*)

Extracts a powerlaw from the data. Does the same as `fit_pwrlaw`, and additionally logs its internal states to stdout.

Parameters

- **timepoints** – The timepoints of the measurements.
- **quantity** – The corresponding quantity, to which a powerlaw-behaviour should be fitted.
- **threshold** – The lower threshold value for the R^2 coefficient of determination. If *threshold* is larger than the R^2 coefficient, a warning is issued and the parameters may be NaN or Inf. Defaults to 0.95. Values above 1 are set to 0.98.

Returns

A 3-tuple with the timepoints, the fitted quantity evaluations and a 4-tuple of the parameters of the following powerlaw: t, (q0, dq_dt, q_trans, n) ↦ q0 + dq_dt * (t - q_trans)**n

ep_bolfi.utility.fitting_functions.fit_sqrt

ep_bolfi.utility.fitting_functions.**fit_sqrt**(*timepoints*, *voltages*, *threshold=0.95*)

See `fit_sqrt_with_warnings` for details. This method does the same, but suppresses inconsequential NumPy warnings.

ep_bolfi.utility.fitting_functions.fit_sqrt_with_warnings

ep_bolfi.utility.fitting_functions.**fit_sqrt_with_warnings**(*timepoints*, *voltages*, *threshold=0.95*)

Extracts a square root at the beginning of the data.

Parameters

- **timepoints** – The timepoints of the measurements.
- **voltages** – The corresponding voltages.
- **threshold** – The lower threshold value for the R^2 coefficient of determination. If *threshold* is smaller than 1, the subset of the experimental data is searched that just fulfills it. Defaults to 0.95. Values above 1 are set to 1.

Returns

A 3-tuple with the timepoints, the fitted voltage evaluations and a 2-tuple of the parameters of the following sqrt function: `t, (U_0, dU_d√t): U_0 + dU_d√t * √(t - timepoints[0])`.

ep_bolfi.utility.fitting_functions.inverse_OCV_fit_function

ep_bolfi.utility.fitting_functions.**inverse_OCV_fit_function**(*SOC*, **args*, *z=1.0*, *T=298.15*, *inverted=True*)

The inverse of `fitting_functions.OCV_fit_function`.

Approximately OCV(SOC). Requires that Δx entries are sorted by x. This corresponds to the parameters being sorted by decreasing E_0.

Parameters

- **E_OCV** – The SOCs for which the voltages shall be evaluated.
- ***args** – A list which length is dividable by three. These are the parameters E_0, a and Δx from "A parametric OCV model" in the order (E_0_0, a_0, Δx_0, E_0_1, a_1, Δx_1…).
- **z** – The charge number of the electrode interface reaction.
- **T** – The temperature of the electrode.
- **inverted** – False uses the formulation from "A parametric OCV model". The default True flips the SOC argument internally to correspond to the more widely adopted convention for the SOC direction.

Returns

The evaluation of the inverse model function.

ep_bolfi.utility.fitting_functions.inverse_d2_dSOC2_OCV_fit_function

ep_bolfi.utility.fitting_functions.**inverse_d2_dSOC2_OCV_fit_function**(*SOC*, **args*, *z=1.0*, *T=298.15*, *inverted=True*)

The 2nd derivative of the inverse of `OCV_fit_function`.

Approximately OCV"(SOC). Requires that Δx entries are sorted by x. This corresponds to the parameters being sorted by decreasing E_0.

Parameters

- **E_OCV** – The SOCs for which the voltages shall be evaluated.
- **args** – A list which length is dividable by three. These are the parameters E_0, a and Δx from "A parametric OCV model" in the order (E_0_0, a_0, Δx_0, E_0_1, a_1, Δx_1…).
- **z** – The charge number of the electrode interface reaction.
- **T** – The temperature of the electrode.
- **inverted** – False uses the formulation from "A parametric OCV model". The default True flips the SOC argument internally to correspond to the more widely adopted convention for the SOC direction.

Returns

The second derivative of the inverse model function.

ep_bolfi.utility.fitting_functions.inverse_d_dSOC_OCV_fit_function

ep_bolfi.utility.fitting_functions.**inverse_d_dSOC_OCV_fit_function**(*SOC*, **args*, *z=1.0*, *T=298.15*, *inverted=True*)

! The derivative of the inverse of `OCV_fit_function`.

Approximately OCV'(SOC). Requires that Δx entries are sorted by x. This corresponds to the parameters being sorted by decreasing E_0.

Parameters

- **E_OCV** – The SOCs for which the voltages shall be evaluated.
- **args** – A list which length is dividable by three. These are the parameters E_0, a and Δx from "A parametric OCV model" in the order (E_0_0, a_0, Δx_0, E_0_1, a_1, Δx_1…).
- **z** – The charge number of the electrode interface reaction.
- **T** – The temperature of the electrode.
- **inverted** – False uses the formulation from "A parametric OCV model". The default True flips the SOC argument internally to correspond to the more widely adopted convention for the SOC direction.

Returns

The evaluation of the derivative of the inverse model function.

ep_bolfi.utility.fitting_functions.laplace_transform

ep_bolfi.utility.fitting_functions.**laplace_transform**(*x*, *y*, *s*)

Performs a basic Laplace transformation.

Parameters

- **x** – The independent variable.
- **y** – The dependent variable.
- **s** – The (possibly complex) frequencies for which to perform the transform.

Returns
The evaluation of the laplace transform at *s*.

ep_bolfi.utility.fitting_functions.smooth_fit

ep_bolfi.utility.fitting_functions.**smooth_fit**(*x*, *y*, *order=3*, *splits=None*, *w=None*, *s=None*, *display=False*, *derivatives=0*)

Calculates a smoothed spline with derivatives.

Note: the `roots` method of a spline only works if it is cubic, i.e. of third order. Each `derivative` reduces the order by one.

Parameters

- **x** – The independent variable.
- **y** – The dependent variable ("plotted over *x*").
- **order** – Interpolation order of the spline.
- **splits** – Optional tuning parameter. A list of points between which splines will be fitted first. The returned spline then is a fit of these individual splines.
- **w** – Optional list of weights. Works best if `1 / w` approximates the standard deviation of the noise in *y* at each point. Defaults to the SciPy default behaviour of `scipy.interpolate.UnivariateSpline`.
- **s** – Optional tuning parameter. Higher values lead to coarser, but more smooth interpolations and vice versa. Defaults to SciPy default behaviour of `scipy.interpolate.UnivariateSpline`.
- **display** – If set to True, the fit parameters of the spline will be printed to console. If possible, a monomial representation is printed.
- **derivatives** – The derivatives of the spline to also include in the return. Default is 0, which gives the spline. 1 would give the spline, followed by its derivative. Can not be higher than spline order. Derivatives are only continuous when `derivatives < order`.

Returns
A smoothing spline in the form of `scipy.UnivariateSpline`.

ep_bolfi.utility.fitting_functions.verbose_spline_parameterization

ep_bolfi.utility.fitting_functions.**verbose_spline_parameterization**(*coeffs*, *knots*, *order*, *format='python'*, *function_name='OCV'*, *function_args='SOC'*, *derivatives=0*, *spline_transformation=''*, *verbose=False*)

Gives the monomic representation of a B-spline.

Parameters

- **coeffs** – The B-spline coefficients as used in `scipy.interpolate`.
- **knots** – The B-spline knots as used in `scipy.interpolate`.
- **order** – The order of the B-spline.
- **format** – Gives the file/language format for the function representation. Default is 'python'. The other choice is 'matlab'.
- **function_name** – An optional name for the printed Python function.
- **function_args** – An optional string for the arguments of the function.
- **derivatives** – The derivatives of the spline to also include in the return. Default is 0, which gives the spline. 1 would give the spline, followed by its derivative. Can not be higher than spline order. Derivatives are only continuous when `derivatives < order`.
- **spline_transformation** – Give a string if you want to include a function that gets applied to the whole spline, e.g. 'exp'. Note that only this one extra case gets handled for the first derivative.
- **verbose** – Print information about the progress of the conversion.

Returns

A string that gives a Python function when "exec"-uted.

Classes

NDArrayEncoder(*[, skipkeys, ensure_ascii, ...])	
OCV_fit_result(fit, SOC, OCV[, SOC_offset, ...])	Contains OCV fit parameters and related information.

ep_bolfi.utility.fitting_functions.NDArrayEncoder

class ep_bolfi.utility.fitting_functions.**NDArrayEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `JSONEncoder`

__init__(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

Methods

__init__(*[, skipkeys, ensure_ascii, ...])	Constructor for JSONEncoder, with sensible defaults.
default(item)	Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).
encode(o)	Return a JSON string representation of a Python data structure.
iterencode(o[, _one_shot])	Encode the given object and yield each string representation as available.

Attributes

item_separator
key_separator

default(*item*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

encode(*o*)

Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*, *_one_shot=False*)

Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

ep_bolfi.utility.fitting_functions.OCV_fit_result

class ep_bolfi.utility.fitting_functions.**OCV_fit_result**(*fit*, *SOC*, *OCV*, *SOC_offset=1.0*, *SOC_scale=1.0*, *optimize_result=None*, *spline_interpolation_knots=None*, *spline_interpolation_coefficients=None*, *function_string=None*)

Bases: `object`

Contains OCV fit parameters and related information.

Reference

C. R. Birkl, E. McTurk, M. R. Roberts, P. G. Bruce and D. A. Howey. "A Parametric Open Circuit Voltage Model for Lithium Ion Batteries". Journal of The Electrochemical Society, 162(12):A2271-A2280, 2015

__init__(*fit*, *SOC*, *OCV*, *SOC_offset=1.0*, *SOC_scale=1.0*, *optimize_result=None*, *spline_interpolation_knots=None*, *spline_interpolation_coefficients=None*, *function_string=None*)

Parameters

fit – The fit parameters of the OCV function from Birkl et al., either with or without the estimated SOC range at the beginning, and optionally without the last Δx entry, which is then fixed to ensure that the sum over Δx is 1. Order of parameters: [E_0_0, a_0, Δx_0, E_0_1, a_1, …].

Methods

SOC_adjusted([soc])	Gives the adjusted SOC values.
SOC_other_electrode([soc])	Relates the SOCs of the two electrodes to each other.
__init__(fit, SOC, OCV[, SOC_offset, ...])	
to_json()	Gives a complete representation in JSON format.

Attributes

SOC_range	The SOC range of the data.
fit	The fit parameters of the OCV function from Birkl et al., excluding the estimated SOC range.
E_0	The E_0 (plateau voltages) parameters.
a	The a (inverse plateau widths) parameters.
Δx	The Δx (phase proportion) parameters.
SOC	The SOC data points.
OCV	The OCV data points.
SOC_offset	If another electrode was factored out in the data, this may contain its SOC at SOC 0 of the electrode of interest.
SOC_scale	If another electrode was factored out in the data, this may contain the the rate of change of its SOC to that of the electrode of interest.
optimize_result	The scipy.optimize.OptimizeResult that led to the fit.
spline_interpolation_knots	The knots of the interpolating spline fitted to the inverse.
spline_interpolation_coefficients	The coefficients of the interpolating spline fitted to the inverse.
function_string	The string representation of the interpolating spline fitted to the inverse.

E_0

The E_0 (plateau voltages) parameters.

OCV

The OCV data points. May be adjusted from the original data.

SOC

The SOC data points.

SOC_adjusted(*soc=None*)

Gives the adjusted SOC values.

Parameters

soc – The SOC as assigned in the original data. This usually corresponds to the range available during a measurement.

Returns

The SOC as corrected by the OCV model. These values will try to correspond to the level of lithiation.

SOC_offset

If another electrode was factored out in the data, this may contain its SOC at SOC 0 of the electrode of interest.

SOC_other_electrode(*soc=None*)

Relates the SOCs of the two electrodes to each other.

If the original data was of a full cell and the other electrode was factored out, this may contain the function that takes the SOC of the electrode of interest and gives the SOC of the other electrode, i.e., the stoichiometric relation.

Parameters
soc – The SOC of the electrode of interest.

Returns
The SOC of the other electrode that was factored out.

SOC_range
The SOC range of the data.

SOC_scale
If another electrode was factored out in the data, this may contain the the rate of change of its SOC to that of the electrode of interest.

a
The a (inverse plateau widths) parameters.

fit
The fit parameters of the OCV function from Birkl et al., excluding the estimated SOC range.

function_string
The string representation of the interpolating spline fitted to the inverse.

optimize_result
The scipy.optimize.OptimizeResult that led to the fit.

spline_interpolation_coefficients
The coefficients of the interpolating spline fitted to the inverse.

spline_interpolation_knots
The knots of the interpolating spline fitted to the inverse.

to_json()
Gives a complete representation in JSON format.

Returns
A JSON-formatted string.

Δx
The Δx (phase proportion) parameters.

ep_bolfi.utility.preprocessing

Contains frequently used workflows in dataset preprocessing.

The functions herein are a collection of simple, but frequent, transformations of arrays of raw measurement data.

Functions

OCV_from_CC_CV(charge, cv, discharge, name, ...)	Tries to extract the OCV curve from CC-CV cycling data.
approximate_confidence_ellipsoid(parameters, ...)	Approximate a confidence ellipsoid.
calculate_SOC(timepoints, currents[, ...])	Transforms applied current over time into SOC.
calculate_both_SOC_from_OCV(parameters, ...)	Calculates the SOC of both electrodes from their OCV.
calculate_desired_voltage(solution, t_eval, ...)	Takes a `pybamm.Solution` object and postprocesses its voltage information, to give overpotentials and/or three-electrode setups.
calculate_means_and_standard_deviations(...)	Calculate means and standard deviations.
capacity(parameters[, electrode])	Convenience function for calculating the capacity.
combine_parameters_to_try(parameters, ...)	Give every combination as full parameter sets.
find_occurrences(sequence, value)	Gives indices in *sequence* where it is closest to *value*.
fix_parameters(parameters_to_be_fixed)	Returns a function which sets some parameters in advance.
laplace_transform(x, y, s)	Performs a basic Laplace transformation.
parallel_simulator_with_setup(model, ...)	A `multiprocessing`-compatible part of `simulate_all_parameter_combinations`.
prepare_parameter_combinations(parameters, ...)	Calculates all permutations of the parameter boundaries.
simulate_all_parameter_combinations(model, ...)	Creates permutations of *parameters_to_try* and simulates them.
solve_all_parameter_combinations(model, ...)	Creates permutations of *parameters_to_try* and solves them.
subtract_OCV_curve_from_cycles(dataset, ...)	Removes the OCV curve from a cycling measurement.
subtract_both_OCV_curves_from_cycles(...[, ...])	Removes the OCV curve from a single cycle.

ep_bolfi.utility.preprocessing.OCV_from_CC_CV

ep_bolfi.utility.preprocessing.**OCV_from_CC_CV**(*charge*, *cv*, *discharge*, *name*, *phases*, *eval_points=200*, *spline_SOC_range=(0.01, 0.99)*, *spline_order=2*, *spline_smoothing=0.002*, *spline_print=None*, *parameters_print=False*)

Tries to extract the OCV curve from CC-CV cycling data.

Parameters

- **charge** – A `Cycling_Information` object containing the constant charge cycle(s). If more than one CC-CV-cycle shall be analyzed, please make sure that the order of this, cv and discharge align.
- **cv** – A `Cycling_Information` object containing the constant voltage part between charge and discharge cycle(s).
- **discharge** – A `Cycling_Information` object containing the constant discharge cycle(s). These occur after each cv cycle.
- **name** – Name of the material for which the CC-CV-cycling was measured.

- **phases** – Number of phases in the `fitting_functions.OCV_fit_function` as an int. The higher it is, the more (over-)fitted the model becomes.
- **eval_points** – The number of points for plotting of the OCV curves.
- **spline_SOC_range** – 2-tuple giving the SOC range in which the inverted fitting_functions.OCV_fit_function will be interpolated by a smoothing spline. Outside of this range the spline is used for extrapolation. Use this to fit the SOC range of interest more precisely, since a fit of the whole range usually fails due to the singularities at SOC 0 and 1. Please note that this range considers the 0-1-range in which the given SOC lies and not the linear transformation of it from the fitting process.
- **spline_order** – Order of this smoothing spline. If it is set to 0, this only calculates and plots the `fitting_functions.OCV_fit_function`.
- **spline_smoothing** – Smoothing factor for this smoothing spline. Default: 2e-3. Lower numbers give more precision, while higher numbers give a simpler spline that smoothes over steep steps in the fitted OCV curve.
- **spline_print** – If set to either 'python' or 'matlab', a string representation of the smoothing spline is printed in the respective format.
- **parameters_print** – Set to True if the fit parameters should be printed to console.

Returns

An 8-tuple consisting of the following: 0: OCV_fits

The fitted OCV curve parameters for each CC-CV cycle as returned by `fitting_functions.fit_OCV`.

1: I_mean
The currents assigned to each CC-CV cycle (without CV).

2: C_charge
The moved capacities during the charge segment(s). This is a list of the same length as charge, cv or discharge.

3: U_charge
The voltages during the charge segment(s). Length: same.

4: C_discharge
The moved capacities during the discharge segment(s). Length: same.

5: U_discharge
The voltages during the discharge segment(s). Length: same.

6: C_evals
Structurally the same as C_charge or C_discharge, this contains the moved capacities that were assigned to the mean voltages of charge and discharge cycle(s).

7: U_means
The mean voltages of each charge and discharge cycle.

ep_bolfi.utility.preprocessing.approximate_confidence_ellipsoid

ep_bolfi.utility.preprocessing.**approximate_confidence_ellipsoid**(*parameters*, *free_parameters_names*, *covariance*, *mean=None*, *transform_parameters={}*, *refinement=True*, *confidence=0.95*)

Approximate a confidence ellipsoid.

Compatible with SubstitutionDict, if *parameters* is one. The geometric approximation is a refinement of the polytope with nodes on the semiaxes of the confidence ellipsoid. The refinement step adds a node for each face, i.e., each sub-polytope with dimension smaller by 1. This node is centered on that face and projected onto the confidence ellipsoid.

Parameters

- **parameters** – The base full parameter set as a dictionary.
- **free_parameters_names** – The names of the parameters that are uncertain as a list. This parameter has to match the order of parameters in *covariance*.
- **covariance** – The covariance of the uncertain parameters as a two-dimensional NumPy array.
- **mean** – The mean of the uncertain parameters as a dictionary. If not set, the values from *parameters* will be used.
- **transform_parameters** – Optional transformations between the parameter space that is used for searching for optimal parameters and the model parameters. Any missing free parameter is not transformed. The values are 2-tuples. The first entry is a function taking the search space parameter and returning the model parameter. The second entry is the inverse function. For convenience, any value may also be one of the following:
  - 'none' => (identity, identity)
  - 'log' => (exp, log)
- **confidence** – The confidence within the ellipsoid. Defaults to 0.95, i.e., the 95% confidence ellipsoid.
- **refinement** – If False, only the nodes on the semiaxes get returned. If True, the nodes centered on the faces get returned as well.

Returns

A 2-tuple where the first item is the list of all parameter set combinations and the second the ellipsoid nodes only as a two-dimensional numpy array with each node in on row.

ep_bolfi.utility.preprocessing.calculate_SOC

ep_bolfi.utility.preprocessing.**calculate_SOC**(*timepoints*, *currents*, *initial_SOC=0*, *sign=1*, *capacity=1*)

Transforms applied current over time into SOC.

Parameters

- **timepoints** – Array of the timepoint segments.
- **currents** – Array of the current segments.

- **initial_SOC** – The SOC value to start accumulating from.
- **sign** – The value by which to multiply the current.
- **capacity** – A scaling by which to convert from C to dimensionless SOC.

Returns
An array of the same shape describing SOC in C.

ep_bolfi.utility.preprocessing.calculate_both_SOC_from_OCV

ep_bolfi.utility.preprocessing.**calculate_both_SOC_from_OCV**(*parameters*, *negative_SOC_from_cell_SOC*, *positive_SOC_from_cell_SOC*, *OCV*)

Calculates the SOC of both electrodes from their OCV.

The SOCs are substitued in the given *parameters*. The SOC of the cell as a whole gets returned in case it is needed.

Parameters

- **parameters** – The parameters of the battery as used for the PyBaMM simulations (see `models.standard_parameters`).
- **negative_SOC_from_cell_SOC** – A function that takes the SOC of the cell and returns the SOC of the negative electrode.
- **positive_SOC_from_cell_SOC** – A function that takes the SOC of the cell and returns the SOC of the positive electrode.
- **OCV** – The OCV for which the SOCs shall be calculated.

Returns
The SOC of the cell as a whole.

ep_bolfi.utility.preprocessing.calculate_desired_voltage

ep_bolfi.utility.preprocessing.**calculate_desired_voltage**(*solution*, *t_eval*, *voltage_scale*, *overpotential*, *three_electrode=None*, *dimensionless_reference_electrode_location=0.5*, *parameters={}*)

Takes a `pybamm.Solution` object and postprocesses its voltage information, to give overpotentials and/or three-electrode setups.

Parameters

- **solution** – The `pybamm.Solution` object to calculate the voltage from.
- **t_eval** – The times at which to evaluate the *solution*.
- **voltage_scale** – The returned voltage gets divided by this value. For example, 1e-3 would produce a plot in [mV], and -1 would flip the sign.
- **overpotential** – If True, only the overpotential of *solutions* gets plotted. Otherwise, the cell voltage (OCV + overpotential) is plotted.

- **three_electrode** – With None, does nothing (i.e., cell potentials are used). If set to either 'positive' or 'negative', instead of cell potentials, the base for the displayed voltage will be the potential of the 'positive' or 'negative' electrode against a reference electrode. For placement of said reference electrode, please refer to *dimensionless_reference_electrode_location*.
- **dimensionless_reference_electrode_location** – The location of the reference electrode, given as a scalar between 0 (placed at the point where negative electrode and separator meet) and 1 (placed at the point where positive electrode and separator meet). Defaults to 0.5 (in the middle).
- **parameters** – The parameter dictionary that was used for the simulation. Only needed for a three-electrode output for the cell geometry.

Returns

The array of the specified voltages over time.

ep_bolfi.utility.preprocessing.calculate_means_and_standard_deviations

ep_bolfi.utility.preprocessing.**calculate_means_and_standard_deviations**(*mean*, *covariance*, *free_parameters_names*, *transform_parameters={}*, *bounds_in_standard_deviations=1*, ***kwargs*)

Calculate means and standard deviations.

Please note that standard deviations translate differently into confidence regions in different dimensions. For the confidence region, use `approximate_confidence_ellipsoid`.

Parameters

- **mean** – The mean of the uncertain parameters as a dictionary.
- **covariance** – The covariance of the uncertain parameters as a two-dimensional NumPy array.
- **free_parameters_names** – The names of the parameters that are uncertain as a list. This parameter maps the order of parameters in *covariance*.
- **transform_parameters** – Optional transformations between the parameter space that is used for searching for optimal parameters and the model parameters. Any missing free parameter is not transformed. The values are 2-tuples. The first entry is a function taking the search space parameter and returning the model parameter. The second entry is the inverse function. For convenience, any value may also be one of the following:
  - 'none' => (identity, identity)
  - 'log' => (exp, log)
- **bounds_in_standard_deviations** – Sets how many standard deviations in each direction the returned error bounds are. These are first applied and then transformed.

- ****kwargs** – Keyword arguments for `scipy.integrate.quad`, which is used to numerically calculate mean and variance.

Returns
A 3-tuple with three dictionaries. Their keys are the free parameters' names as keys and their values are those parameters' means, standard deviations, and error bounds.

ep_bolfi.utility.preprocessing.capacity

`ep_bolfi.utility.preprocessing.`**`capacity`**(*parameters*, *electrode='positive'*)

Convenience function for calculating the capacity.

Parameters

- **parameters** – A parameter file as defined by `models.standard_parameters`.
- **electrode** – The prefix of the electrode to use for capacity calculation. Change to "negative" to use the one with the lower OCP.

Returns
The capacity of the parameterized battery in C.

ep_bolfi.utility.preprocessing.combine_parameters_to_try

`ep_bolfi.utility.preprocessing.`**`combine_parameters_to_try`**(*parameters*, *parameters_to_try_dict*)

Give every combination as full parameter sets.

Compatible with `SubstitutionDict`, if *parameters* is one.

Parameters

- **parameters** – The base full parameter set as a dictionary.
- **parameters_to_try_dict** – The keys of this dictionary correspond to the *parameters*' keys where different values are to be inserted. These are given by the tuples which are the values of this dictionary.

Returns
A 2-tuple where the first item is the list of all parameter set combinations and the second the list of the combinations only.

ep_bolfi.utility.preprocessing.find_occurrences

`ep_bolfi.utility.preprocessing.`**`find_occurrences`**(*sequence*, *value*)

Gives indices in *sequence* where it is closest to *value*.

Parameters

- **sequence** – A list that represents a differentiable function.
- **value** – The value that is searched for in *sequence*. Also, crossings of consecutive values in *sequence* with *value* are searched for.

@return
A list of indices in *sequence* in ascending order where *value* or a close match for *value* was found.

ep_bolfi.utility.preprocessing.fix_parameters

ep_bolfi.utility.preprocessing.**fix_parameters**(*parameters_to_be_fixed*)

Returns a function which sets some parameters in advance.

Parameters

parameters_to_be_fixed – These parameters will at least be a part of the dictionary that the returned function returns.

Returns

The function which adds additional parameters to a dictionary or replaces existing parameters with the new ones.

ep_bolfi.utility.preprocessing.laplace_transform

ep_bolfi.utility.preprocessing.**laplace_transform**(*x*, *y*, *s*)

Performs a basic Laplace transformation.

Parameters

- **x** – The independent variable.
- **y** – The dependent variable.
- **s** – The (possibly complex) frequencies for which to perform the transform.

Returns

The evaluation of the laplace transform at s.

ep_bolfi.utility.preprocessing.parallel_simulator_with_setup

ep_bolfi.utility.preprocessing.**parallel_simulator_with_setup**(*model*, *current_input*, *parameters*, *submesh_types*, *var_pts*, *spatial_methods*, *calc_esoh*, *inputs*, *t_eval*, *voltage_scale*, *overpotential*, *three_electrode*, *dimensionless_reference_electrode_location*, *kwargs*)

A multiprocessing-compatible part of simulate_all_parameter_combinations. See there for details.

ep_bolfi.utility.preprocessing.prepare_parameter_combinations

ep_bolfi.utility.preprocessing.**prepare_parameter_combinations**(*parameters*, *parameters_to_try*, *covariance*, *order_of_parameter_names*, *transform_parameters*, *confidence*)

Calculates all permutations of the parameter boundaries.

Parameters

- **parameters** – The model parameters as a dictionary.
- **parameters_to_try** – A dictionary with the names of the model parameters as keys and lists of the values that are to be tried out for them as values. Mutually exclusive to *covariance*.
- **covariance** – A covariance matrix describing an estimation result of model parameters. Will be used to calculate parameters to try that together approximate the confidence ellipsoid. This confidence ellipsoid will be centered on *parameters*. Mutually exclusive to *parameters_to_try*.
- **order_of_parameter_names** – A list of names from *parameters* that correspond to the order these parameters appear in the rows and columns of *covariance*. Only needed when *covariance* is set.
- **transform_parameters** – Optional transformations between the parameter space that is used for searching for optimal parameters and the model parameters. Any missing free parameter is not transformed. The values are 2-tuples. The first entry is a function taking the search space parameter and returning the model parameter. The second entry is the inverse function. For convenience, any value may also be one of the following:
  - 'none' => (identity, identity)
  - 'log' => (exp, log)
- **confidence** – The confidence within the ellipsoid. Defaults to 0.95, i.e., the 95% confidence ellipsoid.

Returns

A 2-tuple with the individual parameter variations and then all permutations of them.

ep_bolfi.utility.preprocessing.simulate_all_parameter_combinations

ep_bolfi.utility.preprocessing.**simulate_all_parameter_combinations**(*model*, *current_input*, *submesh_types*, *var_pts*, *spatial_methods*, *parameters*, *parameters_to_try=None*, *covariance=None*, *order_of_parameter_names=None*, *additional_input_parameters=[]*, *transform_parameters={}*, *confidence=0.95*, *full_factorial=True*, *calc_esoh=False*, *voltage_scale=1.0*, *overpotential=False*, *three_electrode=None*, *dimensionless_reference_electrode_location=0.5*, *t_eval=None*, ***kwargs*)

Creates permutations of *parameters_to_try* and simulates them.

Parameters

- **model** – The PyBaMM battery model that is to be solved.
- **current_input** – The list of battery operation conditions, in the format of `pybamm.Simulation`.
- **submesh_types** – The submeshes for discretization. See `solversetup.spectral_mesh_pts_and_method`.
- **var_pts** – The number of discretization points. See `solversetup.spectral_mesh_pts_and_method`.
- **spatial_methods** – The spatial methods for discretization. See `solversetup.spectral_mesh_pts_and_method`.
- **parameters** – The model parameters as a dictionary.

- **parameters_to_try** – A dictionary with the names of the model parameters as keys and lists of the values that are to be tried out for them as values. Mutually exclusive to *covariance*.
- **covariance** – A covariance matrix describing an estimation result of model parameters. Will be used to calculate parameters to try that together approximate the confidence ellipsoid. This confidence ellipsoid will be centered on *parameters*. Mutually exclusive to *parameters_to_try*.
- **order_of_parameter_names** – A list of names from *parameters* that correspond to the order these parameters appear in the rows and columns of *covariance*. Only needed when *covariance* is set.
- **additional_input_parameters** – A list of the parameter names that are changed by any of the variable parameters, if *parameters* is a `SubstitutionDict`. Use its `dependent_variables` method to obtain this list.
- **transform_parameters** – Optional transformations between the parameter space that is used for searching for optimal parameters and the model parameters. Any missing free parameter is not transformed. The values are 2-tuples. The first entry is a function taking the search space parameter and returning the model parameter. The second entry is the inverse function. For convenience, any value may also be one of the following:
  - 'none' => (identity, identity)
  - 'log' => (exp, log)
- **confidence** – The confidence within the ellipsoid. Defaults to 0.95, i.e., the 95% confidence ellipsoid.
- **full_factorial** –

  When *parameters_to_try* is set:

  - If True, all parameter combinations are tried out. If False, only each parameter is varied with the others staying fixed.

  When "covariance" is set:

  - If False, only the points on the semiaxes of the confidence ellipsoid constitute the parameters to try. If True, the centres of the faces of the polytope of these points get added to the parameters to try, projected onto the surface of the confidence ellipsoid.
- **calc_esoh** – Passed on to `pybamm.Simulator`, see there.
- **t_eval** – The timepoints at which the "errorbars" shall be evaluated in s. If None are given, the timepoints of the solutions will be chosen.
- ****kwargs** – The optional parameters for `solversetup.solver_setup`.

Returns

A 2-tuple with the model solution for *parameters* as 1st entry. The second entry mimics *parameters_to_try* with each entry in their lists replaced by the model solution for the corresponding parameter substitution. The second entry has one additional key "all parameters", where all *parameters_to_try* combinations are the value.

ep_bolfi.utility.preprocessing.solve_all_parameter_combinations

ep_bolfi.utility.preprocessing.**solve_all_parameter_combinations**(*model*, *t_eval*, *parameters*, *parameters_to_try*, *submesh_types*, *var_pts*, *spatial_methods*, *full_factorial=True*, ***kwargs*)

Creates permutations of *parameters_to_try* and solves them.

Parameters

- **model** – The PyBaMM battery model that is to be solved.
- **t_eval** – The timepoints in s at which this model is to be solved.
- **parameters** – The model parameters as a dictionary.
- **parameters_to_try** – A dictionary with the names of the model parameters as keys and lists of the values that are to be tried out for them as values.
- **submesh_types** – The submeshes for discretization. See `solversetup.spectral_mesh_pts_and_method`.
- **var_pts** – The number of discretization points. See `solversetup.spectral_mesh_pts_and_method`.
- **spatial_methods** – The spatial methods for discretization. See `solversetup.spectral_mesh_pts_and_method`.
- **full_factorial** – If True, all parameter combinations are tried out. If False, only each parameter is varied with the others staying fixed.
- ****kwargs** – The optional parameters for `solversetup.solver_setup`.

Returns

A 2-tuple with the model solution for *parameters* as 1st entry. The second entry mimics *parameters_to_try* with each entry in their lists replaced by the model solution for the corresponding parameter substitution. The second entry has one additional key "all parameters", where all *parameters_to_try* combinations are the value.

ep_bolfi.utility.preprocessing.subtract_OCV_curve_from_cycles

ep_bolfi.utility.preprocessing.**subtract_OCV_curve_from_cycles**(*dataset*, *parameters*, *starting_SOC=None*, *starting_OCV=None*, *electrode='positive'*, *current_sign=0*, *voltage_sign=0*)

Removes the OCV curve from a cycling measurement.

Parameters

- **dataset** – A `Cycling_Information` object of the measurement.
- **parameters** – The parameters of the battery as used for the PyBaMM simulations (see `models.standard_parameters`).

- **starting_SOC** – The SOC at the beginning of the measurement. If not given, the OCV curve will be inverted to determine the initial SOC.
- **starting_OCV** – The OCV at the beginning of the measurement. If not given and *starting_SOC* is also not given, the first entry of voltages is used for this. If not given, but *starting_SOC* is, the OCP function will be evaluated at *starting_SOC* to get the OCV.
- **electrode** – "positive" (default) or "negative" for current sign correction and capacity calculation. "positive" adds SOC with positive current and vice versa. The sign corrections can be overwritten with *x_sign*.
- **current_sign** – 1 adds SOC, -1 subtracts it, 0 follows the default behaviour above.
- **voltage_sign** – 1 subtracts the OCP, -1 adds it, 0 follows the default behaviour above.

Returns

2-tuple. First entry are the voltages minus the OCV as estimated for each data point. These are structured in exactly the same way as in the *dataset*. Second entry are the electrode SOCs as Coulomb-counted in the data.

ep_bolfi.utility.preprocessing.subtract_both_OCV_curves_from_cycles

ep_bolfi.utility.preprocessing.**subtract_both_OCV_curves_from_cycles**(*dataset*, *parameters*, *negative_SOC_from_cell_SOC*, *positive_SOC_from_cell_SOC*, *starting_SOC=None*, *starting_OCV=None*)

Removes the OCV curve from a single cycle.

Parameters

- **dataset** – A `Cycling_Information` object of the measurement.
- **parameters** – The parameters of the battery as used for the PyBaMM simulations (see `models.standard_parameters`).
- **negative_SOC_from_cell_SOC** – A function that takes the SOC of the cell and returns the SOC of the negative electrode.
- **positive_SOC_from_cell_SOC** – A function that takes the SOC of the cell and returns the SOC of the positive electrode.
- **starting_SOC** – The SOC at the beginning of the measurement. If not given, the OCV curves will be inverted to determine the initial SOC.
- **starting_OCV** – The OCV at the beginning of the measurement. If not given, the first entry of voltages is used for this.

Returns

2-tuple. First entry are the voltages minus the OCV as estimated for each data point. These are structured in exactly the same way as in the *dataset*. Second entry are the electrode SOCs as counted in the data.

Classes

SubstitutionDict(storage[, substitutions])	A dictionary with some automatic substitutions.

ep_bolfi.utility.preprocessing.SubstitutionDict

class ep_bolfi.utility.preprocessing.**SubstitutionDict**(*storage*, *substitutions={}*)

Bases: MutableMapping

A dictionary with some automatic substitutions.

substitutions is a dictionary that extends *storage* with automatic substitution rules depending on its value types:

- string, which serves the value of *storage* at that value.
- callable which takes one parameter, which will get passed its SubstitutionDict instance and serves its return value.
- any other type, which serves the value as-is.

Assigning values to keys afterwards will overwrite substitutions.

__init__(*storage*, *substitutions={}*)

Methods

__init__(storage[, substitutions])	
clear()	
dependent_variables(parameters)	Useful to determine the whole amount of effective keys.
get(k[,d])	
items()	
keys()	
log_lock()	Modifies *__get_item__* to log the keys it touched and cleans up the *self._log* after each logging session.
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

clear() → None. Remove all items from D.

dependent_variables(*parameters*)

Useful to determine the whole amount of effective keys.

Parameters

parameters – A list of keys.

Returns
All keys that were touched during substitution rules to produce `[self[p] for p in parameters]`.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

log_lock()
Modifies *__get_item__* to log the keys it touched and cleans up the *self._log* after each logging session.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*,]**F) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

ep_bolfi.utility.visualization

Various helper and plotting functions for common data visualizations.

Functions

bode_plot(fig, ax_real, ax_imag, ω, Z[, ...])	Plot an impedance measurement.
cc_cv_visualization(fig, ax, dataset[, ...])	Automatically labels and displays a CC-CV dataset.
colorline(x, y[, z, cmap, norm, linewidth, ...])	Generates a colored line using `LineCollection`.
fit_and_plot_OCV(ax, SOC, OCV, name, phases)	Fits an SOC(OCV)-model and an OCV(SOC)-evaluable spline.
interactive_impedance_model(frequencies, ...)	Generates a coarse GUI for manual impedance model parameterization.
make_segments(x, y)	Create a list of line segments from *x* and *y* coordinates.
nyquist_plot(fig, ax, ω, Z[, cmap, ls, lw, ...])	Plot an impedance measurement.
plot_ICA(ax, SOC, OCV, name[, spline_order, ...])	Show the derivative of charge by voltage.
plot_OCV_from_CC_CV(ax_ICA_meas, ...[, ...])	Visualizes the `OCV_fitting.OCV_from_CC_CV` output.
plot_comparison(ax, solutions, errorbars, ...)	Tool for comparing simulation and experiment with features.
plot_measurement(fig, ax, dataset, title[, ...])	Plots current and voltage curves in one diagram.
push_apart_text(fig, ax, text_objects[, ...])	Push apart overlapping texts until no overlaps remain.
set_fontsize(ax[, title, xaxis, yaxis, ...])	Convenience function for fontsize changes.
update_legend(ax[, additional_handles, ...])	Makes sure that all items remain and all items show up.
update_limits(ax[, xmin, xmax, ymin, ymax])	Convenience function for adjusting the view.
visualize_correlation(fig, ax, correlation)	Produces a heatmap of a correlation matrix.

ep_bolfi.utility.visualization.bode_plot

ep_bolfi.utility.visualization.**bode_plot**(*fig*, *ax_real*, *ax_imag*, *ω*, *Z*, *cmap=<matplotlib.colors.ListedColormap object>*, *ls_real='-'*, *ls_imag='-.'*, *lw=3*, *title_text='Impedance Measurement'*, *legend_text='impedance'*)

Plot an impedance measurement.

Parameters

- **fig** – The `matplotlib.Figure` for plotting
- **ax** – The `matplotlib.Axes` for plotting.
- **ω** – The frequencies at which the impedeance was measured. May be a list of lists for multiple measurements.
- **Z** – The impedances that were measured at those frequencies. May be a list of lists for multiple measurements.
- **cmap** – The colormap that is used to differentiate multiple impedances.
- **ls_real** – The linestyle of the plot of the real part of the impedance.
- **ls_imag** – The linestyle of the plot of the imaginary part of the impedance.
- **lw** – The linewidth of the plot.
- **title_text** – The text for the title.
- **legend_text** – The text for the legend. May be a list for multiple measurements.

ep_bolfi.utility.visualization.cc_cv_visualization

ep_bolfi.utility.visualization.**cc_cv_visualization**(*fig, ax, dataset, max_number_of_clusters=4, cmap=<matplotlib.colors.ListedColormap object>, check_location=[0.1, 0.7, 0.2, 0.225]*)

Automatically labels and displays a CC-CV dataset.

A checkbutton list gets added for browsing through the labels.

Parameters

- **fig** – The `Figure` where the check boxes shall be drawn.
- **ax** – The `Axes` where the measurements shall be drawn.
- **dataset** – An instance of `Cycling_Information`. Please refer to `utility.dataset_formatting` for further information.
- **max_number_of_clusters** – The maximum number of different labels that shall be tried in the automatic labelling of the dataset.
- **cmap** – The Colormap that is used for colorcoding the cycles.
- **check_location** – The (x,y)-coordinates (first two entries) and the (width,height) (last two entries) of the checkbutton list canvas.

Returns

The CheckButtons instance. The only thing that must be done with this is to keep it in memory. Otherwise, it gets garbage collected ("weak reference") and the checkbuttons don't work.

ep_bolfi.utility.visualization.colorline

ep_bolfi.utility.visualization.**colorline**(*x*, *y*, *z=None*, *cmap=<matplotlib.colors.ListedColormap object>*, *norm=<matplotlib.colors.Normalize object>*, *linewidth=1*, *linestyle='-'*, *alpha=1.0*)

Generates a colored line using `LineCollection`.

http://nbviewer.ipython.org/github/dpsanders/matplotlib-examples/ blob/master/colorline.ipynb http://matplotlib.org/examples/pylab_examples/multicolored_line.html

Parameters

- **x** – The independent variable.
- **y** – The dependent variable.
- **z** – Specify colors.
- **cmap** – Specify a colormap for colors.
- **norm** – Specify a normalization for mapping z to the colormap. Example: `matplotlib.colors.LogNorm(10**(-2), 10**4)`.
- **linewidth** – The linewidth of the generated `LineCollection`.
- **linestyle** – The linestyle of the generated `LineCollection`. If the individual lines in there are too short, its effect might not be visible.
- **alpha** – The transparency of the generated LineCollection.

Returns

A `matplotlib.collections.LineCollection` object *lc*. A Matplotlib axis *ax* can plot it with `ax.add_collection(lc)`.

ep_bolfi.utility.visualization.fit_and_plot_OCV

ep_bolfi.utility.visualization.**fit_and_plot_OCV**(*ax*, *SOC*, *OCV*, *name*, *phases*, *SOC_range_bounds=(0.2, 0.8)*, *SOC_range_limits=(0.0, 1.0)*, *z=1.0*, *T=298.15*, *fit=None*, *eval_SOC=[0, 1]*, *eval_points=200*, *spline_SOC_range=(0.01, 0.99)*, *spline_order=2*, *spline_print=None*, *parameters_print=False*, *inverted=True*, *info_accuracy=True*, *normalized_xaxis=False*, *distance_order=2*, *weights=None*, *initial_parameters=None*, *minimize_options=None*)

Fits an SOC(OCV)-model and an OCV(SOC)-evaluable spline.

Exemplary fit parameters of a graphite anode: `` E_0_g = np.array([0.35973, 0.17454, 0.12454, 0.081957]) γUeminus1_g = np.array([-0.33144, 8.9434e-3, 7.2404e-2, 6.7789e-2]) a_g = a_fit(γUeminus1_g) Δx_g = np.array([8.041e-2, 0.23299, 0.29691, 0.39381])#0.22887 graphite = [p[i] for i in range(4) for p in [E_0_g, a_g, Δx_g]] `` Exemplary fit parameters of an NMC-622 cathode: `` E_0_NMC = np.array([4.2818, 3.9632, 3.9118, 3.6788])

γUeminus1_NMC = np.array([-0.22022, -0.083146, 0.070787, -0.11461]) a_NMC = a_fit(γUeminus1_NMC) Δx_NMC = np.array([0.38646, 0.28229, 0.15104, 0.26562])#0.30105 NMC = [p[i] for i in range(4) for p in [E_0_NMC, a_NMC, Δx_NMC]] ``

Parameters

- **ax** – The `matplotlib.Axes` instance for plotting.
- **SOC** – Presumed SOC points of the OCV measurement. They only need to be precise in respect to relative capacity between measurements. The SOC endpoints of the measurement will be fitted using the `fitting_functions.OCV_fit_function`.
- **OCV** – OCV measurements.
- **name** – Name of the material for which the OCV curve was measured.
- **phases** – Number of phases in the `fitting_functions.OCV_fit_function`. The higher it is, the more (over-)fitted the model becomes.
- **SOC_range_bounds** – Optional hard upper and lower bounds for the SOC correction from the left and the right side, respectively, as a 2-tuple. Use it as a limiting guess for the actual SOC range represented in the measurement. Has to be inside (0.0, 1.0). Set to (0.0, 1.0) to effectively disable SOC range estimation.
- **SOC_range_limits** – Optional hard lower and upper bounds for the SOC correction from the left and the right side, respectively, as a 2-tuple. Use it if you know that your OCV data is incomplete and by how much. Has to be inside (0.0, 1.0). Set to (0.0, 1.0) to allow the SOC range estimation to assign datapoints to the asymptotes.
- **z** – The charge number of the electrode interface reaction.
- **T** – The temperature of the electrode.
- **fit** – May provide the fit parameters if they are already known.
- **eval_SOC** – Denotes the minimum and maximum SOC to plot the OCV curves at.
- **eval_points** – The number of points for plotting of the OCV curves.
- **spline_SOC_range** – 2-tuple giving the SOC range in which the inverted fitting_functions.OCV_fit_function will be interpolated by a smoothing spline. Outside of this range the spline is used for extrapolation. Use this to fit the SOC range of interest more precisely, since a fit of the whole range usually fails due to the singularities at SOC 0 and 1. Please note that this range considers the 0-1-range in which the given SOC lies and not the linear transformation of it from the fitting process.
- **spline_order** – Order of this smoothing spline. If it is set to 0, this only calculates and plots the `fitting_functions.OCV_fit_function`.
- **spline_print** – If set to either 'python' or 'matlab', a string representation of the smoothing spline is printed in the respective format.
- **parameters_print** – Set to True if the fit parameters should be printed to console.
- **inverted** – If True (default), the widely adopted SOC convention is assumed. If False, the formulation of "A parametric OCV model" is used.
- **info_accuracy** – If True, some measures of fit accuracy are displayed in the figure legend: RMSE (root mean square error), MAE (mean absolute error) and ME (maximum error).
- **normalized_xaxis** – If True, the x-axis gets rescaled to [0,1], where {0,1} matches the asymptotes of the OCV fit function.

- **distance_order** – The argument passed to the numpy.linalg.norm of the vector of distances between OCV data and OCV model. Default is 2, i.e., the Euclidean norm. 1 sets it to absolute distance.
- **weights** – Optional weights to apply to the vector of the distances between OCV data and OCV model. Defaults to equal weights.
- **initial_parameters** – Optional initial guess for the model parameters. If left as-is, this will be automatically gleaned from the data. Use only if you have another fit to data of the same electrode material.
- **minimize_options** – Dictionary that gets passed to scipy.optimize.minimize with the method `trust-constr`. See `scipy.optimize.show_options` with the arguments 'minimize' and 'trust-constr' for details.

ep_bolfi.utility.visualization.interactive_impedance_model

ep_bolfi.utility.visualization.**interactive_impedance_model**(*frequencies*, *measured_impedances*, *parameters*, *unknowns*, *transform_unknowns*, *three_electrode=None*, *dimensionless_reference_electrode_location=0.5*, *with_dl_and_sei=False*, *verbose=False*)

Generates a coarse GUI for manual impedance model parameterization.

Parameters

- **frequencies** – The frequencies to plot the model over.
- **measured_impedance** – The impedance to compare against. Has to match *frequencies*.
- **parameters** – The model parameters. See `models.analytic_impedance`.
- **unknowns** – List of parameter names that will be adjustable via sliders.
- **transform_unknowns** – Optional parameter transformations, for e.g. log sliders. Dictionary matching *unknowns*, with values being 2-tuples: the first entry being the slider-to-value transform, and the second entry being the value-to-slider transform.
- **three_electrode** – With None, does nothing (i.e., cell potentials are used). If set to either 'positive' or 'negative', instead of cell potentials, the base for the displayed voltage will be the potential of the 'positive' or 'negative' electrode against a reference electrode. For placement of said reference electrode, please refer to *dimensionless_reference_electrode_location*.
- **dimensionless_reference_electrode_location** – The location of the reference electrode, given as a scalar between 0 (placed at the point where negative electrode and separator meet) and 1 (placed at the point where positive electrode and separator meet). Defaults to 0.5 (in the middle).
- **with_dl_and_sei** – If set to True, the Electrochemical Double Layer and Solid Electrolyte Interphase models in `models.analytic_impedance` get added to the simulation.
- **verbose** – If True, each slider change triggers a log of some characteristic model properties to stdout.

ep_bolfi.utility.visualization.make_segments

ep_bolfi.utility.visualization.**make_segments**(*x*, *y*)

Create a list of line segments from *x* and *y* coordinates.

Parameters

- **x** – The independent variable.
- **y** – The dependent variable.

Returns

An array of the form numlines *x* (points per line) times 2 (*x* and *y*) array. This is the correct format for `matplotlib.collections.LineCollection`.

ep_bolfi.utility.visualization.nyquist_plot

ep_bolfi.utility.visualization.**nyquist_plot**(*fig*, *ax*, *ω*, *Z*, *cmap=<matplotlib.colors.ListedColormap object>*, *ls='-'*, *lw=3*, *alpha=None*, *title_text='Impedance Measurement'*, *legend_text='impedance'*, *colorbar_label='Frequency / Hz'*, *add_frequency_colorbar=True*, *equal_aspect=True*)

Plot an impedance measurement.

Parameters

- **fig** – The `matplotlib.Figure` for plotting
- **ax** – The `matplotlib.Axes` for plotting.
- **ω** – The frequencies at which the impedeance was measured. May be a list of lists for multiple measurements.
- **Z** – The impedances that were measured at those frequencies. May be a list of lists for multiple measurements.
- **cmap** – The colormap that is used to visualize the frequencies.
- **ls** – The linestyle of the plot.
- **lw** – The linewidth of the plot.
- **alpha** – The transparency of the plot. Leave at None for `Impedance_Measurement` plots for automatic shade legend.
- **title_text** – The text for the title.
- **legend_text** – The text for the legend. May be a list for multiple measurements.
- **colorbar_label** – The label that is displayed next to the colorbar.
- **add_frequency_colorbar** – Set to False if *fig* was already decorated with a colorbar.
- **equal_aspect** – Set to False in case of an impedance with extreme aspect ratio.

Returns

A list of the `LineCollection` objects of the impedance plots.

ep_bolfi.utility.visualization.plot_ICA

ep_bolfi.utility.visualization.**plot_ICA**(*ax*, *SOC*, *OCV*, *name*, *spline_order=2*, *spline_smoothing=0.002*, *sign=1*)

Show the derivative of charge by voltage.

Parameters

- **ax** – The `matplotlib.Axes` instance for plotting.
- **SOC** – Presumed SOC points of the OCV measurement. They only need to be precise in respect to relative capacity between measurements.
- **OCV** – OCV measurements as a `list` or `np.array`, matching SOC.
- **name** – Name of the material for which the OCV curve was measured.
- **spline_order** – Order of the smoothing spline used for derivation. Default: 2.
- **spline_smoothing** – Smoothing factor for this smoothing spline. Default: 2e-3. Lower numbers give more precision, while higher numbers give a simpler spline that smoothes over steep steps in the fitted OCV curve.
- **sign** – Put -1 if the ICA comes out negative. Default: 1.

ep_bolfi.utility.visualization.plot_OCV_from_CC_CV

ep_bolfi.utility.visualization.**plot_OCV_from_CC_CV**(*ax_ICA_meas*, *ax_ICA_mean*, *ax_OCV_meas*, *ax_OCV_mean*, *charge*, *cv*, *discharge*, *name*, *phases*, *eval_points=200*, *spline_SOC_range=(0.01, 0.99)*, *spline_order=2*, *spline_smoothing=0.002*, *spline_print=None*, *parameters_print=False*)

Visualizes the `OCV_fitting.OCV_from_CC_CV` output.

Parameters

- **ax_ICA_meas** – The `Axes` where the Incremental Capacity Analysis of the measured charge and discharge cycle(s) shall be plotted.
- **ax_OCV_meas** – The `Axes` where the measured voltage curves shall be plotted.
- **ax_OCV_mean** – The `Axes` where the mean voltage curves shall be plotted.
- **charge** – A `Cycling_Information` object containing the constant charge cycle(s). If more than one CC-CV-cycle shall be analyzed, please make sure that the order of this, *cv*, and *discharge* align.
- **cv** – A `Cycling_Information` object containing the constant voltage part between charge and discharge cycle(s).
- **discharge** – A `Cycling_Information` object containing the constant discharge cycle(s). These occur after each *cv* cycle.
- **name** – Name of the material for which the CC-CV-cycling was measured.
- **phases** – Number of phases in the `fitting_functions.OCV_fit_function`. The higher it is, the more (over-)fitted the model becomes.

- **eval_points** – The number of points for plotting of the OCV curves.
- **spline_SOC_range** – 2-tuple giving the SOC range in which the inverted fitting_functions.OCV_fit_function will be interpolated by a smoothing spline. Outside of this range the spline is used for extrapolation. Use this to fit the SOC range of interest more precisely, since a fit of the whole range usually fails due to the singularities at SOC 0 and 1. Please note that this range considers the 0-1-range in which the given SOC lies and not the linear transformation of it from the fitting process.
- **spline_order** – Order of this smoothing spline. If it is set to 0, this only calculates and plots the `fitting_functions.OCV_fit_function`.
- **spline_smoothing** – Smoothing factor for this smoothing spline. Default: 2e-3. Lower numbers give more precision, while higher numbers give a simpler spline that smoothes over steep steps in the fitted OCV curve.
- **spline_print** – If set to either 'python' or 'matlab', a string representation of the smoothing spline is printed in the respective format.
- **parameters_print** – Set to True if the fit parameters should be printed to stdout.

@param ax_ICA_mean
The `Axes` where the Incremental Capacity Analysis of the mean voltages of charge and discharge cycle(s) shall be plotted.

ep_bolfi.utility.visualization.plot_comparison

ep_bolfi.utility.visualization.**plot_comparison**(*ax*, *solutions*, *errorbars*, *experiment*, *solution_visualization=[]*, *t_eval=None*, *title=''*, *xlabel=''*, *ylabel=''*, *feature_visualizer=<function <lambda>>*, *feature_fontsize=12*, *interactive_plot=False*, *output_variables=None*, *voltage_scale=1.0*, *use_cycles=False*, *overpotential=False*, *three_electrode=None*, *dimensionless_reference_electrode_location=0.5*, *parameters=None*)

Tool for comparing simulation and experiment with features.

First, a `pybamm.QuickPlot` shows the contents of *solutions*. Then, a plot for feature visualization and comparison is generated.

Parameters

- **ax** – The axes onto which the comparison shall be plotted.
- **solutions** – A dictionary of `pybamm.Solution` objects. The key goes into the figure legend and the value gets plotted as a line.
- **errorbars** – A dictionary of lists of either `pybamm.Solution` objects or lists of the desired variable at *t_eval* timepoints. The key goes into the figure legend and the values get plotted as a shaded area between the minimum and maximum.
- **experiment** – A list/tuple of at least length 2. The first two entries are the data timepoints in s and voltages in V. The entries after that are only relevant as optional arguments to *feature_visualizer*.
- **solution_visualization** – This list/tuple is passed on to *feature_visualizer* in place of the additional entries of *experiment* for the visualization of the simulated features.

- **t_eval** – The timepoints at which the *solutions* and *errorbars* shall be evaluated in [s]. If None are given, the timepoints of the *solutions* will be collected instead.
- **title** – The optional title of the feature visualization plot.
- **xlabel** – The optional label of the x-axis there. Please note that the time will be given in [h].
- **ylabel** – The optional label of the y-axis there.
- **feature_visualizer** – This is an optional function that takes *experiment* and returns a list of 2- or 3-tuples. The first two entries in the tuples are x- and y-data to be plotted alongside the other curves. The third entry is a string that is plotted at the respective `(x[0], y[0])`-coordinates.
- **interactive_plot** – Choose whether or not a browsable overview of the solution components shall be shown. Please note that this disrupts the execution of this function until that plot is closed, since it is plotted in a new figure rather than in *ax*.
- **output_variables** – The variables of *solutions* that are to be plotted. When None are specified, some default variables get plotted. The full list of possible variables to plot are returned by PyBaMM models from their `get_fundamental_variables` or `get_coupled_variables` methods. Enter the keys from that as strings in a list here.
- **voltage_scale** – The plotted voltage gets divided by ths value. For example, 1e-3 would produce a plot in [mV]. The voltage given to the *feature_visualizer* is not affected.

:param use_cycles_
If True, the `.cycles` attribute of the *solutions* is used for the *feature_visualizer*. Plotting is not affected.

Parameters

- **overpotential** – If True, only the overpotential of *solutions* gets plotted. Otherwise, the cell voltage (OCV + overpotential) is plotted.
- **three_electrode** – By default, does nothing (i.e., cell potentials are used). If set to either 'positive' or 'negative', instead of cell potentials, the base for the displayed voltage will be the potential of the 'positive' or 'negative' electrode against a reference electrode. For placement of said reference electrode, please refer to *dimensionless_reference_electrode_location*.
- **dimensionless_reference_electrode_location** – The location of the reference electrode, given as a scalar between 0 (placed at the point where negative electrode and separator meet) and 1 (placed at the point where positive electrode and separator meet). Defaults to 0.5 (in the middle).
- **parameters** – The parameter dictionary that was used for the simulation. Only needed for a three-electrode output.

Returns
The text objects that were generated according to *feature_visualizer*.

ep_bolfi.utility.visualization.plot_measurement

ep_bolfi.utility.visualization.**plot_measurement**(*fig*, *ax*, *dataset*, *title*, *cmap=<matplotlib.colors.ListedColormap object>*, *plot_current=True*, *normalize_time=True*)

Plots current and voltage curves in one diagram.

Please don't use `fig.tight_layout()` with this, as it very well might mess up the placement of the colorbar and the second y-axis. Rather, use `plt.subplots(..., constrained_layout=True)`.

Parameters

- **fig** – The `Figure` where the check boxes shall be drawn.
- **ax** – The `Axes` where the measurements shall be drawn.
- **dataset** – An instance of `Cycling_Information`. Please refer to `utility.dataset_formatting` for further information.
- **title** – The optional title of the measurement visualization plot.
- **cmap** – The Colormap that is used for colorcoding the cycles.
- **plot_current** – If True (default), plots the current on a `twinx Axes` as a dashed line. The axis description will be on the right.
- **normalize_time** – If True (default), the timepoints will be changed to start at 0.

Returns

The list of text objects for the numbers.

ep_bolfi.utility.visualization.push_apart_text

ep_bolfi.utility.visualization.**push_apart_text**(*fig*, *ax*, *text_objects*, *lock_xaxis=False*, *temp_path='./temp_render.png'*)

Push apart overlapping texts until no overlaps remain.

Parameters

- **fig** – The figure which contains the text.
- **ax** – The axis which contains the text.
- **text_objects** – A list of the text objects that shall be pushed apart.
- **lock_xaxis** – If True, texts will only be moved in the y-direction.
- **temp_path** – The path to which a temporary image of the figure *fig* gets saved. This is necessary to establish the text bbox sizes.

ep_bolfi.utility.visualization.set_fontsize

ep_bolfi.utility.visualization.**set_fontsize**(*ax*, *title=12*, *xaxis=12*, *yaxis=12*, *xticks=12*, *yticks=12*, *legend=12*)

Convenience function for fontsize changes.

Parameters

- **ax** – The axis which texts shall be adjusted.
- **title** – The new fontsize for the title.
- **xaxis** – The new fontsize for the x-axis label.
- **yaxis** – The new fontsize for the y-axis label.
- **xticks** – The new fontsize for the ticks/numbers at the x-axis.
- **yticks** – The new fontsize for the ticks/numbers at the y-axis.
- **legend** – The new fontsize for the legend entries.

ep_bolfi.utility.visualization.update_legend

ep_bolfi.utility.visualization.**update_legend**(*ax*, *additional_handles=[]*, *additional_labels=[]*, *additional_handler_map={}*)

Makes sure that all items remain and all items show up.

This basically replaces `ax.legend` in a way that makes sure that new items can be added to the legend without losing old ones. Please note that only ***handler_map*s with class keys work correctly.

Parameters

- **ax** – The axis which legend shall be updated.
- **additional_handles** – The same input `ax.legend` would expect. A list of artists.
- **additional_labels** – The same input `ax.legend` would expect. A list of strings.
- **additonal_handler_map** – The same input `ax.legend` would expect for *handler_map*. Please note that, due to the internal structure of the Legend class, only entries with keys that represent classes work right. Entries that have instances of classes (i.e., objects) for keys work exactly once, since the original handle of them is lost in the initialization of a `Legend`.

ep_bolfi.utility.visualization.update_limits

ep_bolfi.utility.visualization.**update_limits**(*ax*, *xmin=inf*, *xmax=-inf*, *ymin=inf*, *ymax=-inf*)

Convenience function for adjusting the view.

Parameters

- **ax** – The axis which viewport shall be adjusted.
- **xmin** – The highest lower bound for the x-axis.
- **xmax** – The lowest upper bound for the x-axis.
- **ymin** – The highest lower bound for the y-axis.
- **ymax** – The lowest upper bound for the y-axis.

ep_bolfi.utility.visualization.visualize_correlation

ep_bolfi.utility.visualization.**visualize_correlation**(*fig*, *ax*, *correlation*, *names=None*, *title=None*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *entry_color='w'*)

Produces a heatmap of a correlation matrix.

Parameters

- **fig** – The `matplotlib.Figure` object for plotting.
- **ax** – The `matplotlib.Axes` object for plotting.
- **correlation** – A two-dimensional (NumPy) array that is the correlation matrix.
- **names** – A list of strings that are names of the variables corresponding to each row or column in the correlation matrix.
- **title** – The title of the heatmap.
- **cmap** – The matplotlib colormap for the heatmap.
- **entry_color** – The colour of the correlation matrix entries.

CHAPTER

THREE

CONTRIBUTING

Contributing to EP-BOLFI

If you'd like to contribute to EP-BOLFI, thank you very much and please have a look at the guidelines below.

Workflow

We use [GIT](https://en.wikipedia.org/wiki/Git) and [GitLab](https://en.wikipedia.org/wiki/GitLab) to coordinate our work. When making any kind of update, we try to follow the procedure below.

Before you begin

1. Create an [issue](https://docs.gitlab.com/ee/user/project/issues/) where new proposals can be discussed before any coding is done.
2. **Download the source code onto your local system, by cloning the repository:**
   `bash git clone https://gitlab.dlr.de/cec/ep-bolfi `
3. **Install the library in editable mode:**
   `bash pip install -e ep-bolfi `
4. **Create a branch of this repo, where all changes will be made, and "checkout" that branch so that your changes live in that branch:**
   `bash git branch <branch_name> git checkout <branch_name> ` Or as a short-hand: `bash git checkout -b <branch_name> `

Writing your code

4. EP-BOLFI is written in [Python](https://en.wikipedia.org/wiki/Python_(programming_language)), and makes heavy use of [ELFI](https://github.com/elfi-dev/elfi) as well as [PyBaMM](https://github.com/pybamm-team/PyBaMM).
5. Make sure to follow our [coding style guidelines](#coding-style-guidelines).
6. Commit your changes to your branch with [useful, descriptive commit messages](https://chris.beams.io/posts/git-commit/): Remember these are visible to all and should still make sense a few months ahead in time. While developing, you can keep using the GitLab issue you're working on as a place for discussion. [Refer to your commits](https://stackoverflow.com/questions/8910271/how-can-i-reference-a-commit-in-an-issue-comment-on-github) when discussing specific lines of code.
7. If you want to add a dependency on another library, or re-use code you found somewhere else, have a look at [these guidelines](#dependencies-and-reusing-code).

Merging your changes with EP-BOLFI

8. Make sure that your code runs successfully. Ideally, implement tests.
9. Run *flake8* on your code to fix formatting issues ahead of time.

10. When you feel your code is finished, or at least warrants serious discussion, create a [merge request](https://docs.gitlab.com/ee/user/project/merge_requests/) (MR) on the [GitLab page of EP-BOLFI](https://gitlab.dlr.de/cec/ep-bolfi).
11. Once a MR has been created, it will be reviewed by any member of the group. Changes might be suggested which you can make by simply adding new commits to the branch. When everything's finished, someone with the right GitLab permissions will merge your changes into the EP-BOLFI main repository.

Coding style guidelines

EP-BOLFI follows the [PEP8 recommendations](https://www.python.org/dev/peps/pep-0008/) for coding style. These are very common guidelines, and community tools have been developed to check how well projects implement them.

Flake8

We use [flake8](http://flake8.pycqa.org/en/latest/) to check our PEP8 adherence. To try this on your system, navigate to the ep_bolfi directory in a terminal and type:

```bash flake8 ```

Documentation

The documentation is generated with [Sphinx](https://www.sphinx-doc.org/) from the source code. This happens automatically during the CI/CD pipeline, with the result being available through GitLab Pages.

Hence, please copy the structure of the in-code documentation for your own comments. It is known as [reStructuredText](https://peps.python.org/pep-0287/).

Dependencies and reusing code

While it's a bad idea for developers to "reinvent the wheel", it's important for users to get a _reasonably_ sized download and an easy install. In addition, external libraries can sometimes cease to be supported, and when they contain bugs it might take a while before fixes become available as automatic downloads to EP-BOLFI users. For these reasons, all dependencies in EP-BOLFI should be thought about carefully, and discussed on GitHub.

Direct inclusion of code from other packages is possible, as long as their license permits it and is compatible with ours, but again should be considered carefully and discussed in the group. Snippets from blogs and [stackoverflow](https://stackoverflow.com/) are often incompatible with other licences than [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) and hence should be avoided. You should attribute (and document) any included code from other packages, by making a comment with a link in the source code.

Building from source

Before pushing your changes, make sure that the version in version.txt is incremented and unique. The CI/CD pipeline will use this to upload a Release with the respective version in each of the kadi tools.

The following instructions are just here to inform you how to package the code yourself if you so desire.

Build and install wheel from source (pip)

Install the build command and execute it: ```bash pip install build python3 -m build ```

The wheel file should be at dist/ep_bolfi-${VERSION}-py3-none-any.whl. Please do not commit these.

Build conda package from wheel (conda)

First build the wheel from source via pip. Then install the necessary packages for building conda packages: ```bash conda install build conda-build conda-verify ```

Then build: ```bash conda-build . ```

The file you need for installing with conda install lies inside your Anaconda distribution, on Windows at conda-bld/win-64/ep_bolfi-${VERSION}-py39_0.tar.bz2.

Building the .xml representations of the kadi tools
```

In order to generate the .xml files for the workflow editor from your version of the kadi_tools, you may use the following on Linux only:

`bash cd ep_bolfi/kadi_tools mkdir xml_representations for file in ./*.py; do if [ $file = "./__init__.py" ]; then continue fi python $file --xmlhelp > xml_representations/${file:2:-3}.xml done `

On Windows, the files get the wrong encoding (UTF-16). If you know what this means, you may generate the .xml files on Windows and manually fix the encoding to be UTF-8.

Please note that, when developing, the version displayed by the workflow editor will always be VERSION, since the actual version is automatically inserted during the CI/CD pipeline.

In the case where spurious lines like "warning in …: failed to import cython module: falling back to numpy" show up, these are due to an unfortunate design decision in GPy. Either delete them manually, or try installing GPy from source like so to improve performance as well:

`bash git clone https://github.com/SheffieldML/GPy.git cd GPy pip install . `

Infrastructure

GitLab

GitLab does some magic with particular filenames. In particular:

- The first page people see when they go to [our GitLab page](https://gitlab.dlr.de/cec/ep-bolfi) displays the contents of README.md, which is written in the [Markdown](https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet) format. Some guidelines can be found [here](https://help.github.com/articles/about-readmes/).
- This file, CONTRIBUTING.md is recognised as the contribution guidelines and a link is [automatically](https://github.com/blog/1184-contributing-guidelines) displayed when new issues or pull requests are created.

Acknowledgements

This CONTRIBUTING.md file was adapted from the excellent [Pints GitHub repo](https://github.com/pints-team/pints).

PYTHON MODULE INDEX

e

INDEX

Non-alphabetical

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z