SimSES

Release 1.2.1

Marc Möller, Daniel Kucevic

Jan 26, 2022

Contents

1 Introduction to SimSES

1.1 Introduction

SimSES (Simulation of stationary energy storage systems) is an open source modeling framework for simulating stationary energy storage systems. The tool, initially developed in MATLAB by Maik Naumann and Nam Truong, was 2019 converted to Python and improved by Marc Moeller and Daniel Kucevic at the Institute for Electrical Energy Storage Technology - Technical University Munich.

SimSES enables a detailed simulation and evaluation of stationary energy storage systems with the current main focus on lithium-ion batteries. Future releases will include redox-flow batteries and Power-to-Gas systems. The main component of the modular and flexible software-tool is an abstract approach to the energy storage model, which allows the variation and hybridization of storage technologies and technical sub-components. Furthermore, stress characterization enables the estimation of the energy storage degradation. Various aging models can be used for this purpose, whereby detailed models based on aging experiments especially for batteries had been developed at the Institute. In order to optimize the utilization of the energy storage in the different applications, a large number of operating strategies are implemented. Time series simulations and built-in evaluations allow to calculate and monitor technical parameters for simulated storage operation. Furthermore, technical and economic key performance indicators (characteristics) are derived and enable the assessment and comparison of the simulation results.

The tool has been used for several publications, including the following papers:

- Kucevic, D.; Tepe, B.; Englberger, S.; Parlikar, A.; Muehlbauer, M.; Bohlen, O.; Jossen, A.; Hesse, H. (2020); Standard Battery Energy Storage System Profiles: Analysis of various Applications for Stationary Lithium-Ion Battery Energy Storage Systems using a Holistic Simulation Framework, doi:10.1016/j.est.2019.101077
- Naumann, M.; Truong, C. N.; Schimpe, M.; Kucevic, D.; Jossen, A.; Hesse, H., "SimSES: Software for techno-economic Simulation of Stationary Energy Storage Systems," International ETG Congress 2017, Bonn, Germany, 2017, pp. 1-6.
- Englberger, S.; Hesse, H.; Kucevic, D.; Jossen, A. A Techno-Economic Analysis of Vehicle-to-Building: Battery Degradation and Efficiency Analysis in the Context of Co-

ordinated Electric Vehicle Charging 2019, 12, doi:10.3390/en12050955.

- Naumann, M.; Karl, R.Ch.; Truong, C.N.; Jossen, A.; Hesse, H.C. (2015): Lithium-ion Battery Cost Analysis in PV-household Application. In: Energy Procedia 73, S. 37-47. DOI: 10.1016/j.egypro.2015.07.555.
- Truong, C.; Naumann, M.; Karl, R.; Mueller, M.; Jossen, A.; Hesse, H. (2016): Economics of Residential Photovoltaic Battery Systems in Germany. The Case of Tesla's Powerwall. In: Batteries 2 (2), S. 14-30. DOI: 10.3390/batteries2020014.

1.2 Quickstart

Note: Installation is only tested on Microsoft Windows 10 Enterprise.

Note: We plan to provide SimSES by PyPi package management in the future

We recommend installing SimSES (and in general third-party) python packages in a virtual environment, encapsulated from the system python distribution. This is optional. To install SimSES in windows, it is currently recommended to use Anaconda and create a virtual environment (Python 3.6 or 3.7) with Anaconda. Creating a virtual environment with Anaconda can be done within the Anaconda Navigator (Environments | Create) or via terminal. A more deailed deocumentaion about virtual environment and how to create them can be found here.

1.2.1 Download and Configure SimSES Project

After creating an virtula environment you can clone SimSES from servers hosted by the 'leibniz rechenzentrum'. You can easily clone (git) SimSES via: https://gitlab.lrz.de/ees-ses/opensimses. After cloning SimSES you can open the project within a Python IDE (e.g. PyCharm). As project interpreter please select the before created virtual environment. In Pycharm this can be done in Settings | ProjectInterpreter.

To start SimSES you have to install some required packages. This can be done by executing setup.py. This installs the following packages:

- scipy
- numpy
- pandas
- matplotlib

SimSES can be simply start with executing the main.py. This script starts the simulation as well as the analysis with some basic settings. All possible settings as well as the code structure will be explained in the following chapter.

1.3 Basic structure of SimSES

SimSES is object-oriented programmed, which allows a modular combination of various components. The core if SimSES is the storage system, which is divided into an AC system and a DC system (see figure *AC storage system and DC Storage system in SimSES.*). Each AC system has one AC/DC converter at least one storage technology and one DC/DC converter. With SimSES it is possible to simulate with more than one storage technology (AC coupled or DC coupled). The selectable components a described in section configuration.

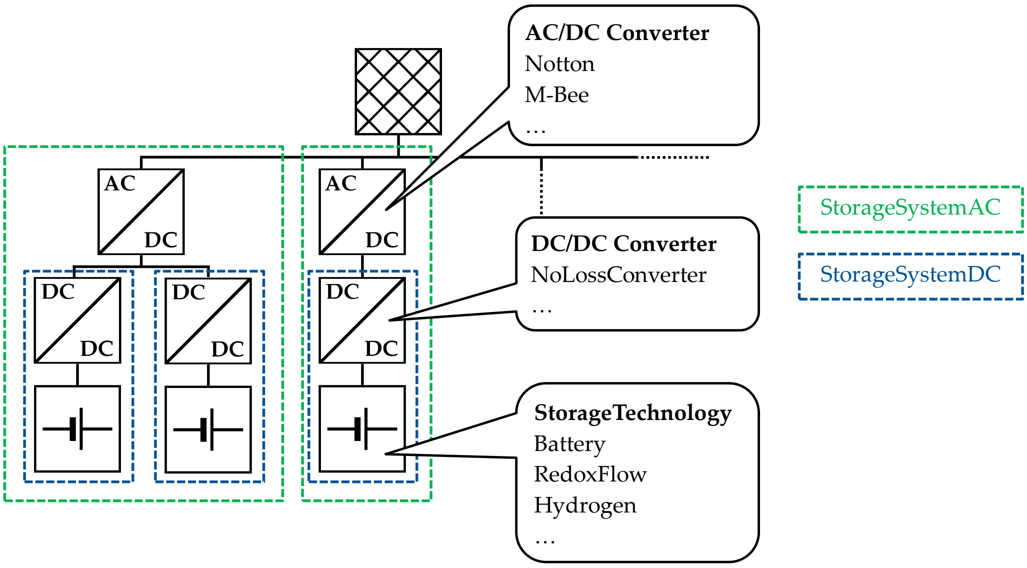

Fig. 1: AC storage system and DC Storage system in SimSES.

1.4 Simulation Loop

SimSES is a time continuous simulation, which means that the calculations are done step by step. The simulation loop is showed in figure *Simulation loop of SimSES*. Based on the selected operation strategy (application) the energy management system (EMS) calculates the AC target power. The AC target power is splitted into the various AC systems. Within each AC system the power is converted by a AC/DC converter and splitted into the various DC systems.

1.5 Configuration

1.6 Contact

simses.ees@ei.tum.de

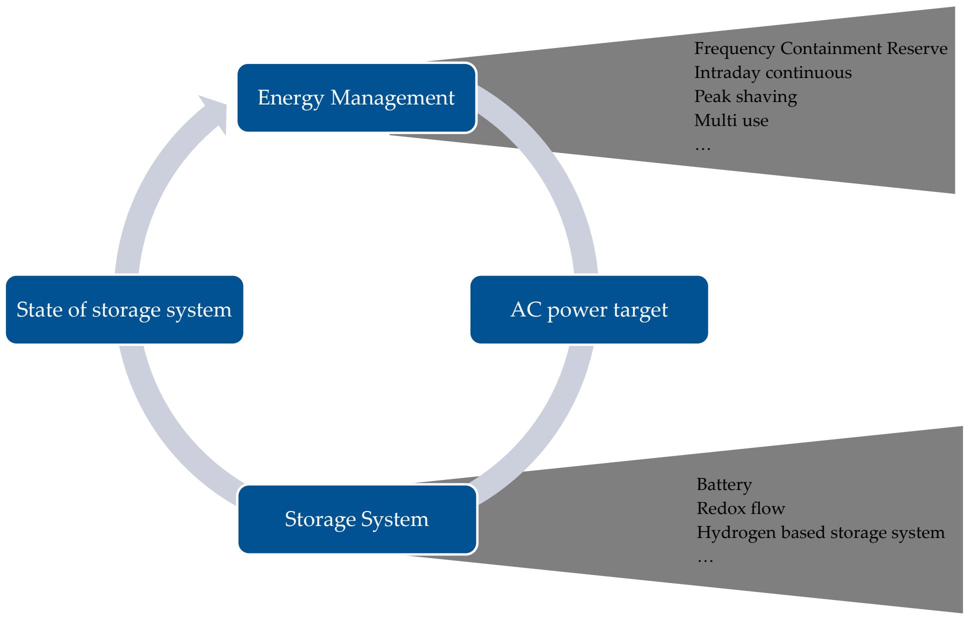

Fig. 2: Simulation loop of SimSES.

2 Code Structure of SimSES

The following chapter shows the code structure of SimSES. This chapter was generated using the programs docstrings.

2.1 Commons

2.1.1 simses.commons package

Subpackages

simses.commons.config package

Subpackages

simses.commons.config.analysis package

Submodules

class AnalysisConfig(*path*, *config*)

Bases: *simses.commons.config.abstract_config.Config*

All analysis configs are inherited from this class

CONFIG_NAME: str = 'analysis'

CONFIG_PATH: str = 'C:\\Users\\kucevic\\Documents\\Python\\opensimses

class EconomicAnalysisConfig(*config*, *path=None*)

Bases: *simses.commons.config.analysis.analysis_config.AnalysisConfig*

Provides

ADD_OPERATION_AND_MAINTENANCE_REVENUE_STREAM: str = 'ADD_OPERATION_AN

```
ANNUAL_RELATIVE_OPERATION_AND_MAINTENANCE_COSTS: str = 'ANNUAL_RELATI
```

```
DEMAND_CHARGE_AVERAGE_INTERVAL: str = 'DEMAND_CHARGE_AVERAGE_INTERVAL
```

```
DEMAND_CHARGE_BILLING_PERIOD: str = 'DEMAND_CHARGE_BILLING_PERIOD'
```

```
DEMAND_CHARGE_PRICE: str = 'DEMAND_CHARGE_PRICE'
```

```
DISCOUNT_RATE: str = 'DISCOUNT_RATE'
```

```
ELECTRICITY_PRICE: str = 'ELECTRICITY_PRICE'
```

```
FCR_PRICE: str = 'FCR_PRICE'
```

```
FCR_USE_PRICE_TIMESERIES: str = 'FCR_USE_PRICE_TIMESERIES'
```

```
IDM_PRICE: str = 'IDM_PRICE'
```

```
IDM_USE_PRICE_TIMESERIES: str = 'IDM_USE_PRICE_TIMESERIES'
```

```
INVESTMENT_COSTS: str = 'INVESTMENT_COSTS'
```

```
PV_FEED_IN_TARIFF: str = 'PV_FEED_IN_TARIFF'
```

```
RENEWABLE_ELECTRICITY_PRICE: str = 'RENEWABLE_ELECTRICITY_PRICE'
```

```
SECTION: str = 'ECONOMIC_ANALYSIS'
```

```
SPECIFIC_INVESTMENT_COSTS_ENERGY: str = 'SPECIFIC_INVESTMENT_COSTS_EN
```

```
SPECIFIC_INVESTMENT_COSTS_POWER: str = 'SPECIFIC_INVESTMENT_COSTS_POW
```

```
USE_SPECIFIC_COSTS: str = 'USE_SPECIFIC_COSTS'
```

property add_o_and_m_revenue_stream
Defines if operation and maintenance revenue stream is used

property annual_realative_o_and_m_costs
Relative annual costs for operation and maintenance related to investment costs in p.u.

property demand_charge_average_interval
Interval length for determining power average in seconds

property demand_charge_billing_period
Defines the billing period. Choose ‘yearly’ or ‘monthly’

property demand_charge_price
Defines the demand charge price in Euro/kW

property discount_rate
Defines the discount rate in p.u.

property electricity_price
Defines the electricity price in Euro/kWh

property fcr_price
FCR price in Euro per kW per day

property fcr_use_price_timeseries
Determine if constant FCR price or prices from FCR_PRICE_PROFILE are used.

property idm_price
IDM price in Euro per kWh

property idm_use_price_timeseries
Determine if constant IDM price or prices from IDM_PRICE_PROFILE are used.

property investment_costs
Defines initial investment costs for the energy storage system

property pv_feed_in_tariff
Defines the feed in tariff in Euro/kWh

property renewable_electricity_price
price for electricity out of renewable source in Euro per kWh

property specific_investment_costs_energy
Defines specific investment costs for the energy storage system in Euro/kWh

property specific_investment_costs_power
Defines additional specific investment costs for the energy storage system in Euro/kW

property use_specific_costs
Determine if specific costs or absolute costs are used to determine the investment costs

class GeneralAnalysisConfig(*config*, *path=None*)
Bases: *simses.commons.config.analysis.analysis_config.AnalysisConfig*

General analysis configs

ECONOMICAL_ANALYSIS: str = 'ECONOMICAL_ANALYSIS'

EXPORT_ANALYSIS_TO_BATCH: str = 'EXPORT_ANALYSIS_TO_BATCH'

EXPORT_ANALYSIS_TO_CSV: str = 'EXPORT_ANALYSIS_TO_CSV'

HYDROGEN_ANALYSIS: str = 'HYDROGEN_ANALYSIS'

LITHIUM_ION_ANALYSIS: str = 'LITHIUM_ION_ANALYSIS'

MERGE_ANALYSIS: str = 'MERGE_ANALYSIS'

PLOTTING: str = 'PLOTTING'

PRINT_RESULTS_TO_CONSOLE: str = 'PRINT_RESULTS_TO_CONSOLE'

REDOX_FLOW_ANALYSIS: str = 'REDOX_FLOW_ANALYSIS'

SECTION: str = 'GENERAL'

SIMULATION: str = 'SIMULATION'

SITE_LEVEL_ANALYSIS: str = 'SITE_LEVEL_ANALYSIS'

SYSTEM_ANALYSIS: str = 'SYSTEM_ANALYSIS'

TECHNICAL_ANALYSIS: str = 'TECHNICAL_ANALYSIS'

property economical_analysis
Returns boolean value for economical analysis directly after the simulation

property export_analysis_to_batch
Defines if analysis results are written to batch files

property export_analysis_to_csv
Defines if analysis results are to be exported to csv files

get_result_for(*path*)
Returns name of the simulation to analyse.

property hydrogen_analysis
Returns boolean value for redox_flow_analysis after the simulation

property lithium_ion_analysis
Returns boolean value for lithium_ion_analysis after the simulation

property logo_file

property merge_analysis
Defines if analysis results are merged

property plotting
Returns boolean value for matplot_plotting after the simulation

property print_result_to_console
Defines if analysis results are to be printed to console

property redox_flow_analysis
Returns boolean value for redox_flow_analysis after the simulation

property site_level_analysis
Returns boolean value for redox_flow_analysis after the simulation

property system_analysis
Returns boolean value for system_analysis after the simulation

property technical_analysis
Returns boolean value for technical analysis directly after the simulation

class MarketProfileConfig(*config*, *path=None*)
Bases: `simses.commons.config.analysis.analysis_config.AnalysisConfig`

Market profile configs

SECTION: str = 'MARKET_PROFILE'

property fcr_price_file
Returns soc profile file_name name from __analysis_config file_name

property intraday_price_file
Return PV generation profile file_name name from __analysis_config file_name

property market_profile_dir
Returns directory of market profiles from __analysis_config file_name

Module contents

simses.commons.config.data package

Submodules

class AuxiliaryDataConfig(*path=None*, *config=None*)
Bases: *simses.commons.config.data.data_config.DataConfig*

property auxiliary_pump_data_dir
Returns directory of pump data files

property pump_eta_file
Returns filename for efficiency file of chosen pump

class BatteryDataConfig(*path=None*, *config=None*)
Bases: *simses.commons.config.data.data_config.DataConfig*

property cell_data_dir
Returns directory of cell data files

property isea_cell_dir
Returns dirname for isea cell files

property lfp_sony_degradation_capacity_file
Returns filename for open circuit voltage of sony LFP

property lfp_sony_degradation_resistance_file
Returns filename for open circuit voltage of sony LFP

property lfp_sony_ocv_file
Returns filename for open circuit voltage of sony LFP

property lfp_sony_rint_file
Returns filename for internal resistance of sony LFP

property lto_lmo_current_file
Returns filename for lto_lmo cell files

property lto_nmc_current_file
Returns filename for lto_nmc cell files

property nca_panasonicNCR_ocv_file
Returns filename for open circuit voltage of panasonic NCA

property nca_panasonicNCR_rint_file
Returns filename for internal resistance of panasonic NCA

property nmc_akasol_akm_ocv_file
Returns filename for open circuit voltage of Akasol-AKM NMC

property nmc_akasol_akm_rint_file
Returns filename for internal resistance of Akasol-AKM NMC

property nmc_akasol_oem_current_file
Returns filename for maximum current of Akasol-OEM NMC

property nmc_akasol_oem_ocv_file
Returns filename for open circuit voltage of Akasol-OEM NMC

property nmc_akasol_oem_rint_file
Returns filename for internal resistance of Akasol-OEM NMC

property nmc_molicel_capacity_cal_file
Returns filename for parameters for calendar aging of NMC Molicel

property nmc_molicel_capacity_cyc_file
Returns filename for parameters for cyclic aging of NMC Molicel

property nmc_molicel_ocv_file
Returns filename for open circuit voltage of NMC Molicel

property nmc_molicel_ri_cal_file
Returns filename for parameters for calendar resistance increase of NMC Molicel

property nmc_molicel_ri_cyc_file
Returns filename for parameters for cyclic resistance increase of NMC Molicel

property nmc_molicel_rint_file
Returns filename for internal resistance of NMC Molicel

property nmc_samsung94test_ocv_file
Returns filename for open circuit voltage of NMC 94Ah SamsungLabTest

property nmc_samsung94test_rint_file
Returns filename for internal resistance of NMC 94Ah SamsungLabTest

property nmc_samsung_120ah_capacity_cal_file
Returns filename for calendar degradation of ads-tec NMC

property nmc_samsung_120ah_ocv_file
Returns filename for open circuit voltage of ads-tec Sanyo NMC

property nmc_samsung_120ah_rint_file
Returns filename for internal resistance of ads-tec NMC

property nmc_sanyo_ocv_file
Returns filename for open circuit voltage of sony Sanyo NMC

property nmc_sanyo_rint_file
Returns filename for internal resistance of Sanyo NMC

property sodium_ion_ocv_green_rock_file
Returns filename for open circuit voltage of sodium-ion Green Rock

property sodium_ion_rint_green_rock_file
Returns filename for internal resistance of sodium_ion Green Rock

class DataConfig(*path*, *config*)

Bases: *simses.commons.config.abstract_config.Config*

DataConfig objects provide information for each specific system path to data

CONFIG_NAME: str = 'data'

CONFIG_PATH: str = 'C:\\Users\\kucevic\\Documents\\Python\\opensimses

class ElectrolyzerDataConfig(*path=None*, *config=None*)

Bases: *simses.commons.config.data.data_config.DataConfig*

property alkaline_electrolyzer_fit_para_file

Returns filename for power curve for PEM-Electrolyzer

property alkaline_electrolyzer_multidim_lookup_currentdensity_file

Returns filename for power curve for Alkaline Electrolyzer

property lookuptable_dir

Returns directory of electrolyzer data files

property parameters_dir

Returns directory of electrolyzer data files

property pem_electrolyzer_multi_dim_analytic_para_file

Returns filename for power curve for PEM-Electrolyzer

property pem_electrolyzer_pc_file

Returns filename for polarisation curve for PEM-Electrolyzer

property pem_electrolyzer_power_file

Returns filename for power curve for PEM-Electrolyzer

class FuelCellDataConfig(*path=None*, *config=None*)

Bases: *simses.commons.config.data.data_config.DataConfig*

property fuel_cell_data_dir

Returns directory of fuelcell data files

property jupiter_fuel_cell_pc_file

Returns filename for polarisation curve for Jupiter-Fuelcell

property jupiter_fuel_cell_power_file

Returns filename for power curve for Jupiter-Fuelcell

property pem_fuel_cell_pc_file

Returns filename for polarisation curve for PEM-Fuelcell

property pem_fuel_cell_power_file

Returns filename for power curve for PEM-Fuelcell

class PowerElectronicsConfig(*path=None*, *config=None*)

Bases: *simses.commons.config.data.data_config.DataConfig*

class top read Power Electronics data path

property acdc_converter_data

Returns directory of acdc converter data files

property aixcontrol_efficiency_file
Returns filename for AixControl GmbH converter

property dcdc_converter_dir
Returns directory of acdc converter data files

property pgs_efficiency_file
Returns filename for Siemens S120 converter

property sinamics_efficiency_file
Returns filename for Siemens S120 converter

class RedoxFlowDataConfig(*path=None*, *config=None*)
Bases: *simses.commons.config.data.data_config.DataConfig*

property cell_stack_shunt_current
Returns filename for shunt current of cell stack 5500W

property high_performance_stack_self_discharge
Returns filename for self-discharge current of high-performance stack 9500W

property high_performance_stack_shunt
Returns filename for shunt current of high-performance stack 9500W

property industrial_stack_1500w_shunt_current
Returns filename for shunt current of industrial stack 1500W

property industrial_stack_9000w_shunt_current
Returns filename for shunt current of industrial stack 9000W

property redox_flow_data_dir
Returns directory of redox flow data files

property redox_flow_hydrogen_evolution_dir
Returns directory of redox flow hydrogen evolution current data files

property rfb_h2_evolution_schweiss_f1_file
Schweiss 2016)

Type Returns filename for the hydrogen evolution of a RFB electrode F1 (source

property rfb_h2_evolution_schweiss_f2_file
Schweiss 2016)

Type Returns filename for the hydrogen evolution of a RFB electrode F2 (source

property rfb_h2_evolution_schweiss_f3_file
Schweiss 2016)

Type Returns filename for the hydrogen evolution of a RFB electrode F3 (source

property rfb_h2_evolution_schweiss_f4_file
Schweiss 2016)

Type Returns filename for the hydrogen evolution of a RFB electrode F4 (source

property rfb_rint_file_cell_stack
Returns filename for internal resistance of the cell stack 5500W

property rfb_rint_file_hp_stack
Returns filename for internal resistance of the high-performance stack 9500W

Module contents

simses.commons.config.generation package

Submodules

class AnalysisConfigGenerator
Bases: *simses.commons.config.generation.generator.ConfigGenerator*

The AnalysisConfigGenerator is a convenience class for generating a config for SimSES analysis. Several options can be activated or deactivated.

do_batch_analysis (*batch=False*)
Multiple simulation results can be merge for comparison of results

Parameters **batch** – True: comparison data is written, default: False

do_data_export (*export=True*)
Analysis results will be written to files

Parameters **export** – True: data is exported, default: True

do_economical_analysis (*analyze=True*)
Provide an economical analysis of the simulation results, e.g., NPV and IRR (only top level system is considered)

Parameters **analyze** – True: Economical analysis is conducted, default: True

do_hydrogen_analysis (*analyze=True*)
Provide a technical analysis for hydrogen systems

Parameters **analyze** – True: Results are analyzed, default: True

do_lithium_ion_analysis (*analyze=True*)
Provide a technical analysis for lithium-ion batteries

Parameters **analyze** – True: Results are analyzed, default: True

do_plotting (*plot=True*)
Figures are created and shown in merged HTML file

Parameters **plot** – True: Figures are plotted, default: True

do_redox_flow_analysis(*analyze=True*)
Provide a technical analysis for redox flow batteries

Parameters **analyze** – True: Results are analyzed, default: True

do_site_level_analysis(*analyze=True*)
Provide a technical analysis on site level, e.g., comparison of site load

Parameters **analyze** – True: Results are analyzed, default: True

do_system_analysis(*analyze=True*)
Provide a technical analysis on system level

Parameters **analyze** – True: Results are analyzed, default: True

do_technical_analysis(*analyze=True*)
Provide a technical analysis of the simulation results, e.g., efficiency and aging

Parameters **analyze** – True: Technical analysis is conducted, default: True

load_default_config()
Loads defaults config

load_local_config()
Loads local config

merge_analysis(*merge=True*)
Analysis results are combined to an HTML file showing results and plots

Parameters **merge** – True: HTML file is written, default: True

print_results(*printing=False*)
Analysis results will be printed to console

Parameters **printing** – True: results are printed, default: False

class ConfigGenerator
Bases: `object`

get_config()

Returns copy of config setup to be passed to SimSES

Return type ConfigParser

load_config_from(*file*)
Loads config from given file

Parameters **file** – path to config file

show()
Printing config setup

class SimulationConfigGenerator
Bases: *simses.commons.config.generation.generator.ConfigGenerator*

The SimulationConfigGenerator is a convenience class for generating a config for a SimSES simulation. Prior knowledge of the options and structure of SimSES is recommended. Before using SimSES within another application it is very helpful to get to know the concepts of SimSES by using it as a standalone tool.

This config generator allows the user to focus only on the options to generate as well as the systems to instantiate without needing to worry about the config structure and naming. However, names of possible classes to instantiate are necessary. Basic implementations like "No"-implementations are provided as convenience methods, other types need to be named directly.

First, you generate options for different kind of components like housing, hvac, acdc / dcdc converter, etc.. For all of these methods you get a return key for this kind of component. This key is needed for the instantiation methods. Second, you define the AC and DC systems with the keys defined and some other values like max. AC power, capacity and intermediate circuit voltage.

You are able to load configs (defaults, local, or your own config file). Please consider that maybe AC and DC systems are already defined which will be instantiated. You can clear these options with the provided clear functions.

`add_acdc_converter`(*converter_type*, *number_of_converters=1*, *switch_value=1.0*)

Adding an acdc converter option to config

Parameters

- **`converter_type`** – examples for ACDC converters: NoLossAcDcConverter, FixEfficiencyAcDcConverter, BonfiglioliAcDcConverter, etc.
- **`number_of_converters`** – possibility to cascade converters with the given number, default: 1
- **`switch_value`** – if converters are cascaded, the switch value in p.u. defines the point of the power to nominal power ratio when the next converter will be activated, default: 1.0

Returns key for acdc converter

Return type str

`add_constant_hvac`(*power*, *temperature*)

Convenience method to add a config option of a constant HVAC system

Parameters

- **`power`** – maximum electrical heating/cooling power in W
- **`temperature`** – set point temperature in centigrade

Returns key for hvac system

Return type str

`add_dcdc_converter`(*converter_type*, *max_power*, *efficiency=None*)

Adding a dcdc converter option to config

Parameters

- **`efficiency`** – efficiency of converter in p.u., only used for FixEfficiencyDcDcConverter
- **`converter_type`** – examples for DCDC converters: NoLossDcDcConverter, FixEfficiencyDcDcConverter, etc.
- **`max_power`** – maximum power of DCDC converter in W

Returns key for dcdc converter

Return type str

`add_fix_efficiency_acdc`()

Convenience method to add a config option of a fix efficiency converter

Returns key for acdc converter

Return type str

`add_fix_efficiency_dcdc`(*efficiency=0.98*)

Convenience method to add a config option of a fix efficiency converter

Parameters **`efficiency`** – efficiency of dcdc converter in p.u.

Returns key for dcdc converter

Return type str

`add_generic_cell`(*capacity*)

Convenience method for constructing a lithium-ion battery with a GenericCell

Parameters **`capacity`** – capacity of battery in Wh

Returns key for storage technology

Return type str

`add_housing`(*housing_type*, *high_cube=False*, *azimuth=0.0*, *absorptivity=0.15*, *ground_albedo=0.2*)

Adding an housing option to config

Parameters

- **`housing_type`** – examples for housing: NoHousing, TwentyFtContainer, etc.
- **`high_cube`** – high cube container are taller than usual containers, default: False
- **`azimuth`** – azimuth angle
- **`absorptivity`** – absorptivity of container
- **`ground_albedo`** – reflection value of ground

Returns key for housing

Return type str

add_hvac(*hvac_type*, *power*, *temperature=25.0*)
Adding an HVAC option to config

Parameters

- **hvac_type** – examples for HVAC: NoHeatingVentilationAirConditioning, FixCOPHeatingVentilationAirConditioning, etc.
- **power** – maximum electrical heating/cooling power in W
- **temperature** – set point temperature in centigrade, default: 25.0

Returns key for hvac system

Return type str

add_hydrogen_technology(*capacity*, *fuel_cell*, *fuel_cell_power*, *electrolyzer*, *electrolyzer_power*, *storage_type*, *pressure*)
Adding a hydrogen energy chain to storage technology options with given parameters.

Parameters

- **capacity** – capacity of battery in Wh
- **fuel_cell** – examples for possible fuel cells: PemFuelCell, JupiterFuelCell
- **fuel_cell_power** – maximum power for fuel cell in W
- **electrolyzer** – examples for possible electrolyzers: PemElectrolyzer, AlkalineElectrolyzer, etc.
- **electrolyzer_power** – maximum power for electrolyzer in W
- **storage_type** – examples for possible storage types: PressureTank, SimplePipeline
- **pressure** – pressure for storage type in bar

Returns key for storage technology

Return type str

add_lithium_ion_battery(*capacity*, *cell_type*, *start_soc=0.5*, *start_soh=1.0*)
Adding a lithium-ion battery to storage technology options with given parameters.

Parameters

- **capacity** – capacity of battery in Wh
- **cell_type** – examples for possible cell types: GenericCell, SonyLFP, PanasonicNCA, MolicelNMC, SanyoNMC
- **start_soc** – state of charge at start of simulation in p.u.
- **start_soh** – state of health at start of simulation in p.u. (Note: not all cell types are supported)

Returns key for storage technology

Return type str

add_no_housing()
Convenience method to add a config option of a no housing

Returns key for housing

Return type str

add_no_hvac()
Convenience method to add a config option of a no HVAC system

Returns key for hvac system

Return type str

add_no_loss_acdc()
Convenience method to add a config option of a no loss converter

Returns key for acdc converter

Return type str

add_no_loss_dcdc()
Convenience method to add a config option of a no loss converter

Returns key for dcdc converter

Return type str

add_redox_flow_battery(*capacity*, *stack_type*, *stack_power*, *pump_algorithm='StoichFlowRate'*)
Adding a redox flow battery to storage technology options with given parameters.

Parameters

- **capacity** – capacity of battery in Wh
- **stack_type** – examples for possible stack types: CellDataStack5500W, DummyStack3000W, IndustrialStack1500W, etc.
- **stack_power** – maximum power of stack in W
- **pump_algorithm** – control algorithm for selected pump, default: StoichFlowRate

Returns key for storage technology

Return type str

add_storage_system_ac(*ac_power*, *intermediate_circuit_voltage*, *acdc_converter*, *housing*, *hvac*)
Adding an AC storage system to the config with the given parameters. All configured system will be instantiated.

Parameters

- **ac_power** – maximum AC power of storage system in W

- **intermediate_circuit_voltage** – voltage of the intermediate circuit in V
- **acdc_converter** – key from generated options for ACDC converter
- **housing** – key from generated options for housing
- **hvac** – key from generated options for hvac

Returns key for AC storage system

Return type str

add_storage_system_dc(*ac_system_name*, *dcdc_converter*, *storage_name*)
Adding an DC storage system to the config with the given parameters. Every DC systems needs to be connected to an AC storage system (via the given key). All configured system will be instantiated.

Parameters

- **ac_system_name** – key from generated options for AC storage systems
- **dcdc_converter** – key from generated options for DCDC converter
- **storage_name** – key from generated options for storage technologies

add_twenty_foot_container()
Convenience method to add a config option of a twenty foot container

Returns key for housing

Return type str

clear_acdc_converter()
Deleting all config options for ACDC converter

clear_dcdc_converter()
Deleting all config options for DCDC converter

clear_housing()
Deleting all config options for housing

clear_hvac()
Deleting all config options for HVAC systems

clear_storage_system_ac()
Deleting all configured AC storage systems

clear_storage_system_dc()
Deleting all configured DC storage systems

clear_storage_technology()
Deleting all configured storage technology options

get_time_format()

Returns expected time format for simulation start and end

Return type str

load_default_config()

Loads defaults config

load_local_config()

Loads local config

no_data_export()

No simulation results will be written to files

set_ambient_temperature_model(*model*, *constant_temperature=None*)

Set the type of ambient temperature model to be used :param constant_temperature: optional parameter for ConstantAmbientTemperature :param model: name of the ambient temperature model :return:

set_ambient_temperature_profile_file(*filename*)

Set the filename for the location ambient temperature profile :param filename: name of specified file as str :return: None

set_battery(*start_soc=None*, *min_soc=None*, *max_soc=None*, *eol=None*, *start_soh=None*, *exact_size='False'*)

Sets parameters for the battery section of the simulation.ini file :param start_soc: SOC at start of simulation :param min_soc: minimum permissible SOC :param max_soc: maximum permissible SOC :param eol: value of SOH at End-of-Life :param start_soh: SOH at start of simulation :param exact_size: Enable or disable rounding of number of cells in parallel/series to reach integer values :return:

set_enable_thermal_simulation(*value*)

Enable/disable thermal simulation :param value: True/False as str :return: None

set_fcr_operation_strategy(*fcr_power*, *idm_power*, *fcr_reserve=0.25*, *soc_set=0.52*)

Setting Frequency Containment Reserve (FCR) including a Intraday Recharge Strategy (IDM) as the operation strategy

Parameters

- **fcr_power** – power to reserve for FCR
- **idm_power** – power to participate in IDM
- **fcr_reserve** – defining the lower and upper bounds for the energy capacity as an equivalent for time with full power in h
- **soc_set** – target value of the SOC considering system losses

set_generation_profile_config(*generation_profile=None*)

Set the filename for the generation profile :param generation_profile: name of specified file as str :return: None

set_ghi_profile_file(*filename*)

Set the filename for the location Global Horizontal Irradiance profile :param filename: name of specified file as str :return: None

set_linear_config(*option=None*, *z_value=None*)
Setting simulation time parameters

Parameters

- **option** –

  examples for current implementations: linear_optimization_efficiency_costs, linear_optimization_calendaric_aging, linear_optimization_and_calendaric_aging

- **z_value** – Z-Value determines the maximum distance between the highest and lowest charged battery storage 0.25 recommendet. Possible values [0 - 1].

set_load_generation_scaling_factor(*load_generation_scaling_factor*)
Sets the generation load scaling factor for the area under the load profile (energy) :param load_generation_scaling_factor: examples: 5e6, 5e7, etc. :type load_generation_scaling_factor: desired load scaling factor as str

set_load_scaling_factor(*load_scaling_factor*)
Sets the load scaling factor for the area under the load profile (energy) :param load_scaling_factor: examples: 5e6, 5e7, etc. :type load_scaling_factor: desired load scaling factor as str

set_operation_strategy(*strategy*, *min_soc=0.0*, *max_soc=1.0*)
Setting the operation strategy

Parameters

- **strategy** – examples for current implementations: PowerFollower, SocFollower, ResidentialPvGreedy, ResidentialPvFeedInDamp, etc.
- **min_soc** – minimum allowed soc of storage technologies considered by the operation strategy
- **max_soc** – maximum allowed soc of storage technologies considered by the operation strategy

set_peak_shaving_strategy(*strategy*, *max_power*)
Setting peak shaving as the operation strategy

Parameters

- **strategy** – examples for current implementations: SimplePeakShaving, PeakShavingPerfectForesight, etc.
- **max_power** – maximum allowed profile power, operations strategy tries to reduce peak power to this value

set_power_distribution_strategy_ac(*strategy*)
Defines the power distribution strategy for AC storage systems

Parameters **strategy** – examples: EqualPowerDistributor, SocBasedPowerDistributor, etc.

set_power_distribution_strategy_dc (*strategy*)
Defines the power distribution strategy for DC storage systems

Parameters **strategy** – examples: EqualPowerDistributor, SocBasedPowerDistributor, etc.

set_profile_config (*load_profile=None*)
Set the filename for the load profile :param load_profile: name of specified file as str :return: None

set_simulation_time (*start=None*, *end=None*, *time_step=60.0*, *loop=1*)
Setting simulation time parameters

Parameters

- **start** – simulation start in expected format
- **end** – simulation start in expected format
- **time_step** – simulation time step in seconds
- **loop** – looping the given simulation time period

set_solar_irradiation_model (*model*)
Set the type of solar irradiation model :param model: name of the ambient temperature model :return: None

Module contents

simses.commons.config.simulation package

Submodules

class BatteryConfig (*config*, *path=None*)
Bases: *simses.commons.config.simulation.simulation_config.SimulationConfig*

Battery specific configs

CALENDAR_LIFETIME: str = 'CALENDAR_LIFETIME'

CELL_PARALLEL_SCALE: str = 'CELL_PARALLEL_SCALE'

CELL_SERIAL_SCALE: str = 'CELL_SERIAL_SCALE'

CONSIDER_VOLTAGE_LIMIT: str = 'CONSIDER_VOLTAGE_LIMIT'

CYCLE_LIFETIME: str = 'CYCLE_LIFETIME'

DEGRADATION_MODEL_NUMBER: str = 'DEGRADATION_MODEL_NUMBER'

EOL: str = 'EOL'

EXACT_SIZE: str = 'EXACT_SIZE'

MAX_SOC: str = 'MAX_SOC'

MIN_SOC: str = 'MIN_SOC'

MULTI_MODEL_PARAMETERS: str = 'MULTI_MODEL_PARAMETERS'

SECTION: str = 'BATTERY'

START_SOC: str = 'START_SOC'

START_SOH: str = 'START_SOH'

START_SOH_SHARE: str = 'START_SOH_SHARE'

property consider_voltage_limit
True

Type Returns if current should be limited by charge / discharge end voltage, default

property degradation_model_number
Number / identifier for selected degradation model for degradation models that read parameters from CSV files (e.g. SonyLFP).

Returns Returns the degradation model parameter number that is to be read from the CSV. Returns 1 if not specified in config.

Return type int

property eol
End of Life criteria (0-1)

Returns Returns EOL criteria in % from data_config file

Return type float

property exact_size
Returns selection for exact sizing True/False

property max_calendar_lifetime
Returns a maximum calendar lifetime for GenericCell degradation in years

property max_equivalent_full_cycles
Returns a number of maximum equivalent full cycles for GenericCell degradation

property max_soc
Maximum SOC (0-1)

Returns Returns the maximum soc from data_config file

Return type float

property min_soc
Minimum SOC (0-1)

Returns Returns the minimum soc from data_config file

Return type float

property multi_model_parameters
Parameters for multi model degradation models. Under development.

Returns Comma seperated parameters in no particular order

Return type str

property parallel_scale
Returns a linear scaling factor of cell in order to simulate a parallel lithium_ion connection

property serial_scale
Returns a linear scaling factor of cell in order to simulate a serial lithium_ion connection

property soc
Minimum SOC (0-1)

Returns Returns the start soc from data_config file

Return type float

property start_soh
End of Life criteria (0-1)

Returns Returns start SOH from data_config file

Return type float

property start_soh_share
Share of start SOH between calendar and cyclic degradation for both, capacity decrease and resistance increase

Returns Returns start SOH share in p.u.

Return type float

class ElectrolyzerConfig(*config*, *path=None*)
Bases: *simses.commons.config.simulation.simulation_config.SimulationConfig*

Electrolyzer specific configs

property desire_pressure_anode
Retruns desired pressure of cathode of electrolyzer

property desire_pressure_cathode
Retruns desired pressure of cathode of electrolyzer

property desire_temperature
Retruns desired pressure of cathode of electrolyzer

property eol
Returns end of life criterion from data_config file_name

class EnergyManagementConfig(*config*, *path=None*)
Bases: *simses.commons.config.simulation.simulation_config.SimulationConfig*

Energy management specific configs

EV_CHARGING_STRATEGY: str = 'EV_CHARGING_STRATEGY'

FCR_RESERVE: str = 'FCR_RESERVE'

MAX_POWER: str = 'MAX_POWER'

MAX_POWER_MONTHLY: str = 'MAX_POWER_MONTHLY'

MAX_POWER_MONTHLY_MODE: str = 'MAX_POWER_MONTHLY_MODE'

MAX_SOC: str = 'MAX_SOC'

MIN_SOC: str = 'MIN_SOC'

POWER_FCR: str = 'POWER_FCR'

POWER_IDM: str = 'POWER_IDM'

SECTION: str = 'ENERGY_MANAGEMENT'

SOC_SET: str = 'SOC_SET'

STRATEGY: str = 'STRATEGY'

property ev_charging_strategy
Returns EV charging strategy from __analysis_config file_name

property fcr_reserve
Returns max soc from __analysis_config file_name

property max_fcr_power
Returns max power for providing frequency containment reserve from __analysis_config file_name

property max_idm_power
Returns max power for intra day market transactions from __analysis_config file_name

property max_power
Returns max power for peak shaving from __analysis_config file_name

property max_power_monthly
Returns a list of monthly max power

property max_power_monthly_mode
Returns max power monthly from __analysis_config file_name

property max_soc
Returns max soc from __analysis_config file_name

property min_soc
Returns min soc from __analysis_config file_name

property operation_strategy
Returns operation strategy from __analysis_config file_name

property soc_set
Returns the optimal soc for a FCR storage from __analysis_config file_name. In case of an overall efficiency below 1, the optimal soc should be higher than 0.5

class FuelCellConfig(*config*, *path=None*)
Bases: `simses.commons.config.simulation.simulation_config.SimulationConfig`

Fuel cell specific configs

property eol
Returns end of life criterion from data_config file_name

property pressure_anode
Retruns desired pressure of cathode of electrolyzer

property pressure_cathode
Retruns desired pressure of cathode of electrolyzer

property temperature
Retruns desired pressure of cathode of electrolyzer

class GeneralSimulationConfig(*config*, *path=None*)
Bases: `simses.commons.config.simulation.simulation_config.SimulationConfig`

General simulation configs

END: str = 'END'

EXPORT_DATA: str = 'EXPORT_DATA'

EXPORT_INTERVAL: str = 'EXPORT_INTERVAL'

LOOP: str = 'LOOP'

SECTION: str = 'GENERAL'

START: str = 'START'

TIME_FORMAT: str = '%Y-%m-%d %H:%M:%S'

TIME_STEP: str = 'TIME_STEP'

property duration
Returns simulation duration in s from __analysis_config file_name

property end
Returns simulation end timestamp

property export_data
Returns selection for data export True/False

property export_interval
Returns interval to write value to file

property loop
Returns number of simulation loops

property start
Returns simulation start timestamp

property timestep
Returns simulation timestep in s

class HydrogenConfig(*config*, *path=None*)
Bases: *simses.commons.config.simulation.simulation_config.SimulationConfig*

Hydrogen storage specific configs

property max_soc
Returns max soc from data_config file_name

property min_soc
Returns min soc from data_config file_name

property soc
Returns start soc from data_config file_name

class ProfileConfig(*config*, *path=None*)
Bases: *simses.commons.config.simulation.simulation_config.SimulationConfig*

Profile specific configs

AMBIENT_TEMPERATURE_PROFILE: str = 'AMBIENT_TEMPERATURE_PROFILE'

BINARY_PROFILE: str = 'BINARY_PROFILE'

FREQUENCY_PROFILE: str = 'FREQUENCY_PROFILE'

GENERATION_PROFILE: str = 'GENERATION_PROFILE'

GENERATION_SCALING_FACTOR: str = 'GENERATION_SCALING_FACTOR'

GLOBAL_HORIZONTAL_IRRADIATION_PROFILE: str = 'GLOBAL_HORIZONTAL_IRRAD

LOAD_FORECAST_PROFILE: str = 'LOAD_FORECAST_PROFILE'

LOAD_PROFILE: str = 'LOAD_PROFILE'

LOAD_SCALING_FACTOR: str = 'LOAD_SCALING_FACTOR'

POWER_PROFILE_DIR: str = 'POWER_PROFILE_DIR'

SCALE_PROFILE_PEAK_POWER: str = 'SCALE_PROFILE_PEAK_POWER'

SECTION: str = 'PROFILE'

SOC_PROFILE: str = 'SOC_PROFILE'

TECHNICAL_PROFILE_DIR: str = 'TECHNICAL_PROFILE_DIR'

THERMAL_PROFILE_DIR: str = 'THERMAL_PROFILE_DIR'

property ambient_temperature_profile_file
Return selected location ambient temperature profile

property binary_profile_file
Return binary profile file_name name from __analysis_config file_name

property frequency_file
Returns frequency profile file_name name from __analysis_config file_name

property generation_profile_file
Return PV generation profile file_name name from __analysis_config file_name

property generation_scaling_factor
Return scaling factor for pv from __analysis_config file_name

property global_horizontal_irradiation_profile_file
Return selected location global horizontal irradiation profile

property load_forecast_file
Returns frequency profile file_name name from __analysis_config file_name

property load_profile
Return selected load profile

property load_scaling_factor
Return scaling factor for the load from __analysis_config file_name

property power_profile_dir
Returns directory of power profiles from __analysis_config file_name

property scale_profile_peak_power

property soc_file
Returns soc profile file_name

property soc_file_value
Returns soc profile value index

property technical_profile_dir
Returns directory of frequency profiles from __analysis_config file_name

property thermal_profile_dir
Returns directory of thermal profiles from __analysis_config file_name

class RedoxFlowConfig(*config*, *path=None*)
Bases: `simses.commons.config.simulation.simulation_config.SimulationConfig`

Redox Flow specific configs

property exact_size
Returns selection for exact sizing True/False

property max_soc
Returns max soc for rfb from __analysis_config file_name

property min_soc
Returns min soc for rfb from __analysis_config file_name

property soc
Returns start soc for rfb from __analysis_config file_name

class SimulationConfig(*path*, *config*)
Bases: `simses.commons.config.abstract_config.Config`

All simulation configs are inherited from this class

CONFIG_NAME: str = 'simulation'

CONFIG_PATH: str = 'C:\\Users\\kucevic\\Documents\\Python\\opensimses

clean_split(*properties*, *delimiter='\n'*)

create_dict_from(*properties*, *delimiter=','*)

create_list_from(*properties*, *delimiter=','*)

class StorageSystemConfig(*config*, *path=None*)
Bases: `simses.commons.config.simulation.simulation_config.SimulationConfig`

Storage system specific configs

ACDC_CONVERTER: str = 'ACDC_CONVERTER'

ACDC_CONVERTER_EFFICIENCY: int = 3

ACDC_CONVERTER_NUMBERS: int = 1

ACDC_CONVERTER_SWITCH: int = 2

ACDC_CONVERTER_TYPE: int = 0

AC_SYSTEM_CONVERTER: int = 3

AC_SYSTEM_DC_VOLTAGE: int = 2

AC_SYSTEM_HOUSING: int = 4

AC_SYSTEM_HVAC: int = 5

AC_SYSTEM_NAME: int = 0

AC_SYSTEM_POWER: int = 1

AMBIENT_TEMPERATURE_CONSTANT: int = 1

AMBIENT_TEMPERATURE_MODEL: str = 'AMBIENT_TEMPERATURE_MODEL'

AMBIENT_TEMPERATURE_TYPE: int = 0

BATTERY_CELL: int = 2

BATTERY_SOC: int = 3

BATTERY_SOH: int = 4

CYCLE_DETECTOR: str = 'CYCLE_DETECTOR'

DCDC_CONVERTER: str = 'DCDC_CONVERTER'

```
DCDC_CONVERTER_EFFICIENCY: int = 2

DCDC_CONVERTER_POWER: int = 1

DCDC_CONVERTER_TYPE: int = 0

DC_POWER_DISTRIBUTOR_TYPE: int = 0

DC_SYSTEM_CONVERTER: int = 1

DC_SYSTEM_NAME: int = 0

DC_SYSTEM_STORAGE: int = 2

ELECTROLYZER_POWER: int = 5

ELECTROLYZER_TYPE: int = 4

FUEL_CELL_POWER: int = 3

FUEL_CELL_TYPE: int = 2

HOUSING: str = 'HOUSING'

HOUSING_ABSORPTIVITY = 3

HOUSING_AZIMUTH: int = 2

HOUSING_GROUND_ALBEDO = 4

HOUSING_HIGH_CUBE: int = 1

HOUSING_TYPE: int = 0

HVAC: str = 'HVAC'

HVAC_KD_COEFFICIENT: int = 5

HVAC_KI_COEFFICIENT: int = 4

HVAC_KP_COEFFICIENT: int = 3

HVAC_POWER: int = 1

HVAC_TEMPERATURE_SETPOINT: int = 2

HVAC_TYPE: int = 0

HYDROGEN_STORAGE: int = 6

HYDROGEN_TANK_PRESSURE: int = 7

LINEAR_DISTRIBUTION_Z_VALUE: str = 'LINEAR_DISTRIBUTION_Z_VALUE'

LINEAR_OPTION: str = 'LINEAR_OPTION'

POWER_DISTRIBUTOR_AC: str = 'POWER_DISTRIBUTOR_AC'

POWER_DISTRIBUTOR_DC: str = 'POWER_DISTRIBUTOR_DC'

REDOX_FLOW_PUMP_ALGORITHM: int = 4

REDOX_FLOW_STACK: int = 2
```

SECTION: str = 'STORAGE_SYSTEM'

SOLAR_IRRADIATION_MODEL: str = 'SOLAR_IRRADIATION_MODEL'

SOLAR_IRRADIATION_TYPE: int = 0

STACK_MODULE_POWER: int = 3

STORAGE_CAPACITY: int = 0

STORAGE_SYSTEM_AC: str = 'STORAGE_SYSTEM_AC'

STORAGE_SYSTEM_DC: str = 'STORAGE_SYSTEM_DC'

STORAGE_TECHNOLOGY: str = 'STORAGE_TECHNOLOGY'

STORAGE_TYPE: int = 1

THERMAL_SIMULATION: str = 'THERMAL_SIMULATION'

property acdc_converter
Returns a list of acdc converter

property ambient_temperature_model
Returns name of ambient temperature model

property cycle_detector
Returns name of cycle detector

property dcdc_converter
Returns a list of acdc converter

property housing
Returns a list of housing objects

property hvac
Returns a list of hvac systems

property linear_distributor_z_value
Returns name of cycle detector

property linear_option
Returns name of cycle detector

property power_distributor_ac
Returns name of cycle detector

property power_distributor_dc
Returns name of cycle detector

property solar_irradiation_model
Returns name of solar irradiation model

property storage_systems_ac
Returns a list of ac storage systems

property storage_systems_dc
Returns a list of dc storage systems

property storage_technologies
Returns a list of storage technologies

property thermal_simulation
Returns user preference to enable/disable thermal simulation

Module contents

Submodules

class Config(*path*, *name*, *config*)
Bases: `abc.ABC`

The Config class contains all necessary configuration options of a package, e.g. simulation or analysis. In addition, the Config class selects proper options from defaults, local or in code configurations. For all sections of each config a own config class is provided and is inherited from Config.

Configs are taken from INI files in config package. *.defaults.ini are read in first, followed by *.local.ini and finally overwritten by a ConfigParser passed in as a constructor argument. This threefold delivers functionality by default if you just want to run a possible setup. If a specific simulation should be run locally, the config parameters of the *.local.ini file will overwrite the configuration of defaults; only necessary parameters can be included in the local config file. Furthermore, for sensitivity analysis the ConfigParser argument is taken into account to automate various configurations. The ConfigParser should have the same structure as the config files and overwrites the given values.

CONFIG_EXT: str = '.ini'

DEFAULTS: str = '.defaults.ini'

LOCAL: str = '.local.ini'

get_data_path(*path*)

get_property(*section*, *option*)
Returns the value for given section and option

Parameters

- **section** – section of config
- **option** – option of config

write_config_to(*path*)
Write current config to a file in given path

Parameters path – directory in which config file should be written

class LogConfig(*path=None*)
Bases: *simses.commons.config.abstract_config.Config*

CONFIG_NAME: str = 'logger'

CONFIG_PATH: str = 'C:\\Users\\kucevic\\Documents\\Python\\opensimses

property log_level
Returns log level

property log_to_console

property log_to_file

classmethod set_config(*config*)

Module contents

simses.commons.cycle_detection package

Submodules

class CycleDetector
Bases: `abc.ABC`

abstract close()
Closing all resources in calendar degradation model

abstract cycle_detected(*time*, *state*)
Cycle Detector. Returns true if the sign (charging to discharge and vice versa) changes or the SOC reaches the maximum or minimum or the end of the simulation is reached

Parameters

- **time** (*current time of the simulation*) –
- **state** (LithiumIonState, *current BatteryState of the lithium_ion state*) –

Returns returns true if a cycle is detected

Return type bool

abstract get_crate()
Determines the mean c-rate of a detected cycle

Returns Mean c-rate of a detected cycle in 1/s

Return type float

abstract get_delta_full_equivalent_cycle()
Determines the delta in full equivalent cycles [0,1]

Returns Delta in full equivalent cycles in p.u.

Return type float

abstract get_depth_of_cycle()
Determines the depth of a detected cycle

Returns Depth of a detected cycle in p.u.

Return type float

abstract get_full_equivalent_cycle()
Determines the total number of full equivalent cycles

Returns Total number of full equivalent cycles

Return type float

abstract get_mean_soc()
Determines the mean SOC of a detected cycle

Returns Mean SOC of a detected cycle in p.u.

Return type float

abstract reset()
resets all values within a degradation model, if battery is replaced

class HalfCycleDetector(*start_soc*, *general_config*)
Bases: *simses.commons.cycle_detection.cycle_detector.CycleDetector*

close()

cycle_detected(*time*, *state*)

get_crate()

get_delta_full_equivalent_cycle()

get_depth_of_cycle()

get_full_equivalent_cycle()

get_mean_soc()

reset()

class NoCycleDetector
Bases: *simses.commons.cycle_detection.cycle_detector.CycleDetector*

close()

cycle_detected(*time*, *state*)

get_crate()

get_delta_full_equivalent_cycle()

get_depth_of_cycle()

get_full_equivalent_cycle()

get_mean_soc()

reset()

Module contents

simses.commons.data package

Submodules

class CSVDataHandler(*result_dir, config, export_cls=<class 'simses.commons.state.abstract_state.State'>*)

Bases: `threading.Thread`, *`simses.commons.data.data_handler.DataHandler`*

Export for Battery data during the simulation into one CSV file_name. Batteries send their parameters for every timestamp to this class, which adds them to the CSV file_name. For Multiprocessing a queue is used.

COMP: str = 'gzip'

EXT_COMP: str = '.gz'

EXT_CSV: str = '.csv'

USE_GZIP_COMPRESSION: bool = True

close()

Closing all open resources in data export

copy_results_to_destination()

Transfer the resulting csv file_name as well as the configuration file_name to the destination folder defined in the class creation.

classmethod get_data_from(*path, state_cls*)

Reads file for Class in path as pandas dataframe

Parameters

- **path** (*Path of simulation results*) –
- **state_cls** (*State class*) –

Returns Data of file in path

Return type pandas.Dataframe

run()

Function to write data from queue to csv.

simulation_done()

Function to let the DataExport Thread know that the simulation is done and stop after emptying the queue.

transfer_data(*data*)

Function to place data into the queue. This data is picked up by the run function and written into a CSV file_name.

Parameters **data** (*list*) –

class DataHandler

Bases: `abc.ABC`

Abstract Base Class for Data Export

abstract close()

Closing all open resources in data export

is_alive()

Checks if a thread is running

abstract simulation_done()

Function to let the DataExport Thread know that the simulation is done and stop after emptying the queue.

abstract start()

Function to write data from queue to csv.

abstract transfer_data(*data*)

Function to place data into the queue. This data is picked up by the run function and written into a CSV file_name.

Parameters data (*list*) –

class NoDataHandler

Bases: *simses.commons.data.data_handler.DataHandler*

Does not export anything.

close()

Closing all open resources in data export

simulation_done()

Function to let the DataExport Thread know that the simulation is done and stop after emptying the queue.

start()

Function to write data from queue to csv.

transfer_data(*data*)

Function to place data into the queue. This data is picked up by the run function and written into a CSV file_name.

Parameters data (*list*) –

Module contents

simses.commons.profile package

Subpackages

simses.commons.profile.economic package

Submodules

class ConstantPrice(*price*)

Bases: `simses.commons.profile.economic.market.MarketProfile`

close()

initialize_profile()

next(*time*)

profile_data_to_list(*sign_factor=1*)

class FcrMarketProfile(*general_config, config*)

Bases: `simses.commons.profile.economic.market.MarketProfile`

Provides the FCR market prices

close()

initialize_profile()

next(*time*)

class IntradayMarketProfile(*general_config, config*)

Bases: `simses.commons.profile.economic.market.MarketProfile`

Provides the intraday market prices

close()

initialize_profile()

next(*time*)

class MarketProfile

Bases: `abc.ABC`

Profile of market prices for a time frame.

abstract close()

Closing all open resources in market profile

abstract initialize_profile()

abstract next(*time*)

provides the next market data (price) of a specific market

Returns next price signal

Return type float

Module contents

simses.commons.profile.power package

Submodules

class AlternatePowerProfile(*power_on=1500.0, power_off=0.0, scaling_factor=1.0, time_on=1.0, time_off=1.0*)

Bases: `simses.commons.profile.power.power_profile.PowerProfile`

AlternatePowerProfile produces an alternating power profile, especially useful for clear defined cycle tests.

close()

next(*time*)

class ConstantPowerProfile(*power=0.0, scaling_factor=1.0*)

Bases: `simses.commons.profile.power.power_profile.PowerProfile`

ConstantPowerProfile delivers a constant power value over time.

close()

next(*time*)

class FilePowerProfile(*config, filename, delimiter=',', scaling_factor=1.0*)

Bases: `simses.commons.profile.power.power_profile.PowerProfile`

FilePowerProfile is a generic implementation of a PowerProfile. It provides power values from a file using FileProfile library.

class Header

Bases: `object`

ANNUAL_CONSUMPTION: str = 'Annual load consumption in kWh'

PEAK_POWER: str = 'Nominal power in kWp'

close()

next(*time*)

profile_data_to_list(*sign_factor=1*)

Extracts the whole time series as a list and resets the pointer of the (internal) file afterwards

Parameters **sign_factor** –

Returns profile values as a list

Return type list

class GenerationProfile(*profile_config*, *general_config*)

Bases: `simses.commons.profile.power.file.FilePowerProfile`

GenerationProfile is a specific implementation of FilePowerProfile. It reads a file with a power and time series. Values with a positive sign are recognized as a generation for the system.

GenerationProfile require a specific header at the top of the file:

Nominal power in kWp: [Value]

PEAK_POWER: str = 'Nominal power in kWp'

Header of generation profile for scaling power values

class LoadProfile(*profile_config*, *general_config*)

Bases: `simses.commons.profile.power.file.FilePowerProfile`

LoadProfile is a specific implementation of FilePowerProfile. It reads a file with a power and time series. Values with a positive sign are recognized as a load for the system.

LoadProfiles require a specific header at the top of the file:

Annual load consumption in kWh: [Value]

ANNUAL_CONSUMPTION: str = 'Annual load consumption in kWh'

Header of load profile for scaling power values

class PowerProfile

Bases: `abc.ABC`

Power Profile for the energy management system of the ESS.

abstract close()

Closing all open resources in power profile

abstract next(*time*)

provides the power for the next step of a specific load or generator (e.g. Household load profile)

Parameters **time** (*current timestamp of the simulation*) –

Returns power for the next step

Return type float

class RandomPowerProfile(*start_time*, *max_power=1500.0*, *power_offset=0.0*, *scaling_factor=1.0*, *time_increment=1.0*)

Bases: `simses.commons.profile.power.power_profile.PowerProfile`

RandomPowerProfile is a specific implementation of PowerProfile. It delivers a random power value for each timestep. Power series are reproducable - the used Random class uses the a specific seed for calculating random numbers.

close()

next (*time*)

Module contents

simses.commons.profile.technical package

Submodules

class FrequencyProfile (*config, profile_config*)

Bases: `simses.commons.profile.technical.technical.TechnicalProfile`

close ()

next (*time*)

class SocProfile (*config, profile_config*)

Bases: `simses.commons.profile.technical.technical.TechnicalProfile`

close ()

next (*time*)

class TechnicalProfile

Bases: `abc.ABC`

Profile for additional data, e.g. frequency.

abstract close ()

Closing all open resources in market profile

abstract next (*time*)

provides the data for a technical profile (SoC or frequency)

Returns next data in a specific profile

Return type float

Module contents

Submodules

class FileProfile (*config, filename, delimiter=', ', scaling_factor=1, value_index=1, interpolation=<simses.commons.timeseries.interpolation.linear_interpolation.LinearInterp object>, average=<simses.commons.timeseries.average.mean_average.MeanAver object>*)

Bases: `object`

FileProfile is able to read time series from a file. It supports several unit and date formats, takes timezones into consideration (default: Berlin) and interpolates respectively averages values of the given time series. If no time series is given, a time generator generates the timestep according to the given sampling rate or simulation timestep.

A header could inherit the given timezone, value unit, time unit and sampling rate:

Unit: W

Timezone: Berlin

Time: s

Sampling in s: 3600s

class Header
Bases: `object`

LATITUDE: str = 'Latitude'

LONGITUDE: str = 'Longitude'

SAMPLING: str = 'Sampling in s'

TIME: str = 'Time'

TIMEZONE: str = 'Timezone'

UNIT: str = 'Unit'

close()

classmethod get_header_from(*filename*)
Extracts header from given file

Attention: Only searches in the first ten lines for a header!

Parameters **filename** –

Returns header with key/value pairs

Return type dict

get_latitude()

Returns Latitude value of profile location, raises an exception if not available

Return type float

get_longitude()

Returns Longitude value of profile location, raises an exception if not available

Return type float

classmethod get_max_value_of(*filename, delimiter=',', value_idx=1, start=0.0*)
Searching for maximum value within given file

Attention: This method could be very time consuming depending on the profile to read.

Parameters

- **filename** – filename of profile which is read in
- **delimiter** – delimiter for values in one line, default: ','
- **value_idx** – column number of line for value which should be considered, default: 1
- **start** – start value for search

Returns maximum value of given profile

Return type float

get_timezone()

Returns timezone for profile location, raises an exception if not available

Return type float

initialize_file()

Allows to re-initialize the profile to start reading values from the beginning again

next(*time*)

Retrieves the next value for given time from file time series

Parameters **time** – time as epoch timestamp

Returns (interpolated/averaged) value for given time

Return type float

classmethod open_file(*filename*)

Opens file with filename depending on the file type. Currently .gz and .csv are supported. The file type extension is added if necessary.

Parameters **filename** – path to file

Returns

Return type file object (needs to be closed manually)

profile_data_to_list(*sign_factor=1*)

Extracts the whole time series as a list and resets the pointer of the (internal) file afterwards

Parameters **sign_factor** –

Returns profile values as a list

Return type list

Module contents

simses.commons.state package

Subpackages

simses.commons.state.technology package

Submodules

class ElectrolyzerState(*system_id*, *storage_id*)

Bases: *simses.commons.state.technology.storage.StorageTechnologyState*

Current physical state of the electrolyzer components with the main electrical parameters.

CONVECTION_HEAT = 'convection heat in W'

CURRENT = 'current of electrolyzer in A'

CURRENT_DENSITY = 'current density of electrolyzer in A'

EXCHANGE_CURRENT_DENS_DECREASE = 'decrease j0 in p.u.'

EXCHANGE_CURRENT_DENS_DECREASE_CALENDAR = 'decrease j0 calendric in p

EXCHANGE_CURRENT_DENS_DECREASE_CYCLIC = 'decrease j0 cyclic in p.u.'

FULFILLMENT = 'fulfillment in p.u.'

HYDROGEN_OUTFLOW = 'hydrogen outlfow in mol/s'

HYDROGEN_PRODUCTION = 'hydrogen production in mol/s'

MAX_CHARGE_POWER = 'Maximum charging power in W'

MAX_DISCHARGE_POWER = 'Maximum discharging power in W'

OXYGEN_OUTFLOW = 'oxygen outflow in mol/s'

OXYGEN_PRODUCTION = 'oxygen production in mol/s'

PART_PRESSURE_H2 = 'partial pressure H2 in bar'

PART_PRESSURE_O2 = 'partial pressure O2 in bar'

POWER = 'power in W'

POWER_COMPRESSOR = 'power for compression of hydrogen in W'

POWER_GAS_DRYING = 'power for drying of hydrogen in W'

POWER_LOSS = 'Power loss in W'

POWER_PUMP = 'power water circulation electrolyzer in W'

```
POWER_WATER_HEATING = 'power water heating electrolyzer in W'
PRESSURE_ANODE = 'relative anode pressure of electrolyzer in barg'
PRESSURE_CATHODE = 'relative cathode pressure or electrolyzer in barg
REFERENCE_VOLTAGE = 'reverence voltage in V'
RESISTANCE_INCREASE = 'increase R total in p.u.'
RESISTANCE_INCREASE_CALENDAR = 'increase reisistance calendric in p.u
RESISTANCE_INCREASE_CYCLIC = 'increase resistance cyclic in p.u.'
SAT_PRESSURE_H2O = 'saturation pressure H2O in bar'
SOH = 'SOH electrolyzer in p.u.'
SYSTEM_AC_ID = 'StorageSystemAC'
SYSTEM_DC_ID = 'StorageSystemDC'
TEMPERATURE = 'temperature of electrolyzer stack in K'
TOTAL_HYDROGEN_PRODUCTION = 'total amount of produced hydrogen in kg'
VOLTAGE = 'voltage of electrolyzer in V'
WATER_FLOW = 'waterflow electrolyzer in mol/s'
WATER_OUTFLOW_ANODE = 'watersteam outflow anode electrolyzer in mol/s
WATER_OUTFLOW_CATHODE = 'watersteam outflow cathode electrolyzer in m
WATER_USE = 'wateruse in mol/s'
property capacity
property convection_heat
property current
property current_density
property exchange_current_decrease
property exchange_current_decrease_calendar
property exchange_current_decrease_cyclic
property fulfillment
property hydrogen_outflow
property hydrogen_production
property id
property is_charge
property max_charge_power
property max_discharge_power
```

property oxygen_outflow

property oxygen_production

property part_pressure_h2

property part_pressure_o2

property power

property power_compressor

property power_gas_drying

property power_loss

property power_pump

property power_water_heating

property pressure_anode

property pressure_cathode

property reference_voltage

property resistance_increase

property resistance_increase_calendar

property resistance_increase_cyclic

property sat_pressure_h2o

property soc

property soh

classmethod sum_parallel(*hydrogen_states*)

classmethod sum_serial(*states*)

property temperature

property total_hydrogen_production

property voltage

property water_flow

property water_outflow_anode

property water_outflow_cathode

property water_use

class FuelCellState(*system_id*, *storage_id*)

Bases: `simses.commons.state.technology.storage.StorageTechnologyState`

Current physical state of the fuel cell components with the main electrical parameters.

CONVECTION_HEAT = 'convection heat in W'

CURRENT = 'current of fuel cell in A'

CURRENT_DENSITY = 'current density of fuel cell in A cm-2'

FULFILLMENT = 'fulfillment in p.u.'

HYDROGEN_INFLOW = 'hydrogen inflow in mol/s'

HYDROGEN_USE = 'hydrogen use in mol/s'

MAX_CHARGE_POWER = 'max charge power in W'

MAX_DISCHARGE_POWER = 'max discharge power in W'

OXYGEN_INFLOW = 'oxygen inflow in mol/s'

OXYGEN_USE = 'oxygen use in mol/s'

POWER = 'power in W'

POWER_LOSS = 'Power loss in W'

PRESSURE_ANODE = 'relative anode pressure of fuelcell in barg'

PRESSURE_CATHODE = 'relative cathode pressure of fuelcell in barg'

SOC = 'SOC of Hydrogen Storage'

SOH = 'State of health in p.u.'

SYSTEM_AC_ID = 'StorageSystemAC'

SYSTEM_DC_ID = 'StorageSystemDC'

TEMPERATURE = 'temperature of fuel cell stack in K'

VOLTAGE = 'voltage of fuel cell in V'

property capacity

property convection_heat

property current

property current_density

property fulfillment

property hydrogen_inflow

property hydrogen_use

property id

property is_charge

property max_charge_power

property max_discharge_power

property oxygen_inflow

property oxygen_use

property power_loss

property pressure_anode

property pressure_cathode

property soc

property soh

classmethod sum_parallel(*hydrogen_states*)

classmethod sum_serial(*states*)

property temperature

property voltage

class HydrogenState(*system_id*, *storage_id*)

Bases: `simses.commons.state.technology.storage.StorageTechnologyState`

Current physical state of the hydrogen storage components with the main electrical parameters.

CAPACITY: str = 'capacity in Wh'

CURRENT: str = 'current of hydrogen storage system in A'

FULFILLMENT: str = 'fulfillment in p.u.'

MAX_CHARGE_POWER = 'Maximum charging power in W'

MAX_DISCHARGE_POWER = 'Maximum discharging power in W'

POWER: str = 'Power in W'

POWER_LOSS: str = 'Power loss in W'

SOC: str = 'SOC in p.u.'

SOH = 'State of health in p.u.'

SYSTEM_AC_ID: str = 'StorageSystemAC'

SYSTEM_DC_ID: str = 'StorageSystemDC'

TEMPERATURE: str = 'temperature in K'

VOLTAGE: str = 'voltage of hydrogen system in V'

property capacity

property current

property fulfillment

property id

property is_charge

property max_charge_power

property max_discharge_power

property power

property power_loss

property soc

property soh

classmethod sum_parallel (*states*)

classmethod sum_serial (*states*)

property temperature

property voltage

class LithiumIonState (*system_id, storage_id*)

Bases: *simses.commons.state.technology.storage.StorageTechnologyState*

Current physical state of the lithium_ion with the main electrical parameters.

CAPACITY: str = 'Capacity in Wh'

CAPACITY_LOSS_CALENDRIC: str = 'Calendric Capacity Loss in Wh'

CAPACITY_LOSS_CYCLIC: str = 'Cyclic Capacity Loss in Wh'

CAPACITY_LOSS_OTHER: str = 'Other Capacity Loss in Wh'

CURRENT: str = 'Current in A'

FULFILLMENT: str = 'Bat_ful in p.u.'

INTERNAL_RESISTANCE: str = 'Internal resistance in Ohm'

MAX_CHARGE_POWER: str = 'Maximum charging power in W'

MAX_DISCHARGE_POWER: str = 'Maximum discharging power in W'

NOMINAL_VOLTAGE: str = 'Nominal voltage in V'

POWER_LOSS: str = 'P_loss in W'

RESISTANCE_INCREASE: str = 'R increase in p.u.'

RESISTANCE_INCREASE_CALENDRIC: str = 'Calendric R Increase in p.u.'

RESISTANCE_INCREASE_CYCLIC: str = 'Cyclic R Increase in p.u.'

RESISTANCE_INCREASE_OTHER: str = 'Other R Increase in p.u.'

SOC: str = 'SOC in p.u.'

SOE: str = 'State of energy in Wh'

SOH: str = 'State of health in p.u.'

SYSTEM_AC_ID: str = 'StorageSystemAC'

SYSTEM_DC_ID: str = 'StorageSystemDC'

TEMPERATURE: str = 'Temperature in K'

VOLTAGE: str = 'Voltage in V'

VOLTAGE_OPEN_CIRCUIT: str = 'Open circuit voltage in V'

property capacity

property capacity_loss_calendric

property capacity_loss_cyclic

property capacity_loss_other

property current

property fulfillment

property id

property internal_resistance

property is_charge

property max_charge_power

property max_discharge_power

property nominal_voltage

property power_loss

property resistance_increase

property resistance_increase_calendric

property resistance_increase_cyclic

property resistance_increase_other

property soc

property soe

property soh

classmethod sum_parallel(*battery_states*)

Classmethod to calculate the combined BatteryState over several batteries connected in parallel.

Parameters battery_states (*List*) – List of BatteryState which are connected in parallel.

Returns Combined BatteryState over the parallel BatteryStates.

Return type battery_state

classmethod sum_serial(*battery_states*)

Classmethod to calculate the combined battery_state over several batteries connected in serial.

Parameters **battery_states** (*List*) – List of serial connected battery_states.

Returns Combined battery_state over the serial connected battery_states.

Return type battery_state

property temperature

property voltage

property voltage_open_circuit

class RedoxFlowState(*system_id*, *storage_id*)

Bases: *simses.commons.state.technology.storage.StorageTechnologyState*

Current physical state of the redox_flow system with the main electrical parameters.

CAPACITY = 'Capacity in Wh'

CURRENT = 'Current in A'

FLOW_RATE_ANOLYTE = 'Anolyte flow rate in m^3/s'

FLOW_RATE_CATHOLYTE = 'Catholyte flow rate in m^3/s'

FULFILLMENT = 'Fulfillment in p.u.'

INTERNAL_RESISTANCE = 'Internal resistance in Ohm'

MAX_CHARGE_POWER = 'Maximum charging power in W'

MAX_DISCHARGE_POWER = 'Maximum discharging power in W'

OPEN_CIRCUIT_VOLTAGE = 'Open circuit voltage (OCV) in V'

POWER = 'Power output in W'

POWER_IN = 'Power input in W'

POWER_LOSS = 'Power loss in W'

PRESSURE_DROP_ANOLYTE = 'Anolyte pressure drop in Pa'

PRESSURE_DROP_CATHOLYTE = 'Catholyte pressure drop in Pa'

PRESSURE_LOSS_ANOLYTE = 'Pressure loss anolyte in W'

PRESSURE_LOSS_CATHOLYTE = 'Pressure loss catholyte in W'

PUMP_POWER = 'Pump power in W'

SOC = 'SOC in p.u.'

SOC_STACK = 'SOC in stack in p.u.'

SOH = 'State of health in p.u.'

SYSTEM_AC_ID = 'StorageSystemAC'

SYSTEM_DC_ID = 'StorageSystemDC'

TEMPERATURE = 'Electrolyte and stack temperature in K'

TIME_PUMP = 'Time of pump on or off in s'

VOLTAGE = 'Voltage in V'

property capacity

property current

property flow_rate_anolyte

property flow_rate_catholyte

property fulfillment

property id

property internal_resistance

property is_charge

property max_charge_power

property max_discharge_power

property open_circuit_voltage

property power

property power_in

property power_loss

property pressure_drop_anolyte

property pressure_drop_catholyte

property pressure_loss_anolyte

property pressure_loss_catholyte

property pump_power

property soc

property soc_stack

property soh

classmethod sum_parallel(*states*)

classmethod sum_serial(*states*)

property temperature

property time_pump

property voltage

class StorageTechnologyState

Bases: `simses.commons.state.abstract_state.State`, `abc.ABC`

A state with information specifically provided by a storage technology for each simulation timestep.

abstract property capacity
Capacity in Wh

abstract property current
DC Current in A

abstract property fulfillment
Stage of power fulfillment of storage data in p.u.

abstract property is_charge
Returns True if current state presents a charging process, false otherwise

abstract property max_charge_power
Current maximum charging power of storage technology in W

abstract property max_discharge_power
Current maximum discharging power of storage technology in W

abstract property power_loss
Power losses in W

abstract property soc
SOC in p.u.

abstract property soh
SOC in p.u.

abstract property temperature
Temperature of storage technology in K

abstract property voltage
DC Voltage in V

Module contents

Submodules

class State
Bases: `abc.ABC`

Basic abstract class for all states in SimSES. States in SimSES are date objects 1) transfering data from one object to another and 2) documenting results of simulation. Those written state files are used for the analysis of the simulation. There are no cached results in SimSES, all states are written immediately after each simulation timestep. The underlying state object in this class is a plain dictonary.

TIME = 'Time in s'

add (*state*)
Adding all items of state to self.state

Parameters **state** (State) –

Returns

Return type None

divide_by (*divisor*, *key=None*)

Divide all values of self.state by divisor. If a key is provided, only the value for the key is divided.

Parameters

- **divisor** (*float*) –
- **key** (*str*) –

Returns

Return type None

get (*key*)

Returns value for key

Parameters **key** (*str*) –

Returns

Return type value as float

classmethod header ()

Returns a list of all state keys

abstract property id

init_variable (*key*)

Intitalize variable of key with 0.

Parameters **key** (*str*) –

Returns

Return type None

set (*key*, *value*)

Sets value to variable of key

Parameters

- **key** (*str*) –
- **value** (*float*) –

Returns

Return type None

set_all (*state*)

Setting all items of state to self.state

Parameters **state** (State) –

Returns

Return type None

abstract classmethod sum_parallel(*states*)
Classmethod to calculate the combined state over several states connected in parallel.

Parameters **states** (*List*) – List of State which are connected in parallel.

Returns Combined State over the parallel States.

Return type state

classmethod sum_reverse(*key*, *states*)
Calculation the inverted sum of inverse values (like parallel resistances).

Parameters

- **key** (*str*) – Key of the state.
- **states** (*list*) – List of states.

Returns Summed up value.

Return type float

abstract classmethod sum_serial(*states*)
Classmethod to calculate the combined state over several states connected in serial.

Parameters **states** (*List*) – List of State which are connected in serial.

Returns Combined State over the serial States.

Return type state

property time
Time in s

to_export()
Returns a dictionary with a list of all values

Returns values combined with state.

Return type dict

property to_list
Returns a list of all values of self.state

Returns

Return type list

class EnergyManagementState
Bases: *simses.commons.state.abstract_state.State*

Current State of the Energy Management (PV, Load, etc..)

FCR_MAX_POWER = 'Power reserved for FCR in W'

IDM_POWER = 'Power delivered for IDM in W'

LOAD_POWER = 'Load in W'

PEAKSHAVING_LIMIT = 'Peak Shaving Limit in W'

PV_POWER = 'PV Generation in W'

property fcr_max_power

property id

property idm_power

property load_power

property peakshaving_limit

property pv_power

classmethod sum_parallel(*system_states*)

classmethod sum_serial(*states*)

class SystemParameters

Bases: `object`

SystemParameters collects all static information of the storage system and writes it to the results folder.

ACDC_CONVERTER: str = 'acdc_converter'

AUXILIARIES: str = 'auxiliaries'

BATTERIES: str = 'batteries'

BATTERY_CIRCUIT: str = 'battery_circuit'

CELL_TYPE: str = 'cell_type'

CONTAINER_NUMBER: str = 'number_of_containers'

CONTAINER_TYPE: str = 'container_type'

DCDC_CONVERTER: str = 'dcdc_converter'

EXTENSION: str = '.txt'

ID: str = 'id'

NOMINAL_VOLTAGE: str = 'nominal_voltage'

PARAMETERS: str = 'parameters'

POWER_DISTRIBUTION: str = 'power_distribution'

SECTION: str = 'System'

STORAGE_TECHNOLOGY: str = 'technology'

SUBSYSTEM: str = 'subsystems'

SYSTEM: str = 'system'

classmethod get_file_name()

Returns file name of system parameters

Return type str

set(*parameter*, *value*)
Setting a parameters value

Parameters

- **parameter** – Parameter provided by SystemParameter class
- **value** – value to be written to parameter as string

set_all(*parameters*)
Setting all parameters to internal system parameters

Parameters parameters – a set of parameters written to system parameters

write_parameters_to(*path*)
Writing system parameters to file in path

Parameters path – path of file to write parameters

class SystemState(*system_id*, *storage_id*)
Bases: *simses.commons.state.abstract_state.State*

Current physical state of the system with the main electrical parameters.

AC_POWER = 'AC power in W'

AC_POWER_DELIVERED = 'AC_P_delivered in W'

AMBIENT_TEMPERATURE = 'Ambient temperature in K'

AUX_LOSSES = 'Aux losses in W'

CAPACITY = 'Capacity in Wh'

DC_CURRENT = 'DC current in A'

DC_POWER_ADDITIONAL = 'DC power additional in W'

DC_POWER_INTERMEDIATE_CIRCUIT = 'DC power of intermediate circuit in

DC_POWER_LOSS = 'DC power loss in W'

DC_POWER_STORAGE = 'DC power of storage in W'

DC_VOLTAGE_CIRCUIT = 'DC voltage of intermediate circuit in V'

FULFILLMENT = 'Fulfillment in p.u.'

HVAC_THERMAL_POWER = 'HVAC thermal power in W'

MAX_CHARGE_POWER = 'Maximum charging power in W'

MAX_DISCHARGE_POWER = 'Maximum discharging power in W'

PE_LOSSES = 'PE_losses in W'

SOC = 'SOC in p.u.'

SOH = 'State of Health in p.u.'

SOLAR_THERMAL_LOAD = 'Solar irradiation thermal load in W'

STORAGE_POWER_LOSS = 'DC power loss of storage technology in W'

SYSTEM_AC_ID = 'StorageSystemAC'

SYSTEM_DC_ID = 'StorageSystemDC'

TEMPERATURE = 'Internal air temperature in K'

property ac_power

property ac_power_delivered

property ambient_temperature

property aux_losses

property capacity

property dc_circuit_voltage

property dc_current

property dc_power_additional

property dc_power_intermediate_circuit

property dc_power_loss

property dc_power_storage

property fulfillment

property hvac_thermal_power

property id

property is_charge

property max_charge_power

property max_discharge_power

property pe_losses

property soc

property soh

property solar_thermal_load

property storage_power_loss

classmethod sum_parallel(*system_states*)

classmethod sum_serial(*states*)

property temperature

Module contents

simses.commons.test package

Submodules

test_next (*result*, *uut*)

uut (*power*, *scaling_factor*)

create_config ()

create_file (*value*)

test_next (*time_factor*, *result*, *uut*)

uut (*power*, *scaling_factor*)

test_next (*time*, *recent_time*, *recent_value*, *last_time*, *last_value*, *result*, *uut*)

uut ()

create_list_with (*start*, *end*)

test_next (*start*, *end*, *result*, *uut*)

uut ()

Module contents

simses.commons.timeseries package

Subpackages

simses.commons.timeseries.average package

Submodules

class Average

Bases: `abc.ABC`

Averages values of multiple TimeValues with a specific behaviour (arithmetic mean, median, . . .)

abstract average (*data*)

Averages values of data

Parameters data – List of TimeValue objects

Returns averaged value

Return type float

class MeanAverage

Bases: *simses.commons.timeseries.average.average.Average*

MeanAverage provides a arithmetic mean behaviour of averaging.

average(*data*)

Module contents

simses.commons.timeseries.interpolation package

Submodules

class Interpolation

Bases: abc.ABC

Interpolation implementations are to interpolate values between values as TimeValue objects. Each implementation has a specific interpolation behaviour (linear, splines, . . .).

abstract close()

Closing resources in interpolation

abstract interpolate(*time*, *recent*, *last*)

Interpolate values of given TimeValue objects to time

Parameters

- **time** – timestamp in s which lies in between recent and last
- **recent** – TimeValue with a timestamp >= time
- **last** – TimeValue with a timestamp <= time

Returns interpolated value

Return type float

static is_necessary(*tstmp*, *data*)

class LastValue

Bases: *simses.commons.timeseries.interpolation.interpolation.Interpolation*

Returns last value of given time range

close()

interpolate(*time*, *recent*, *last*)

class LinearInterpolation

Bases: *simses.commons.timeseries.interpolation.interpolation.Interpolation*

Linear interpolation behaviour

close()

interpolate(*time*, *recent*, *last*)

Module contents

Submodules

class TimeValue(*tstmp*, *value*)

Bases: `object`

TimeValue is a data class with a timestamp connected to value.

static sort_by_time(*data*, *descending=False*)

In-place sorting of list with TimeValue objects by time (ascending)

Parameters

- **descending** – reverse sorting of data list, default: False
- **data** – list of TimeValue objects

property time

property value

Module contents

simses.commons.utils package

Submodules

create_line_from(*name*, *values*)

count_loc(*lines*)

count_loc_in_directory(*directory*, *ext*, *start=''*, *consider_comments=False*)

annotate_heatmap(*im*, *data=None*, *valfmt='{x:.2f}'*, *textcolors=['black', 'white']*, *threshold=None*, ***textkw*)

A function to annotate a heatmap.

Parameters

- **im** – The AxesImage to be labeled.

- **data** – Data used to annotate. If None, the image's data is used. Optional.
- **valfmt** – The format of the annotations inside the heatmap. This should either use the string format method, e.g. "$ {x:.2f}", or be a *matplotlib.ticker.Formatter*. Optional.
- **textcolors** – A list or array of two color specifications. The first is used for values below a threshold, the second for those above. Optional.
- **threshold** – Value in data units according to which the colors from textcolors are applied. If None (the default) uses the middle of the colormap as separation. Optional.
- ****kwargs** – All other arguments are forwarded to each call to *text* used to create the text labels.

heatmap(*data*, *row_labels*, *col_labels*, *ax=None*, *cbar_kw={}*, *cbarlabel=''*, ***kwargs*)

Create a heatmap from a numpy array and two lists of labels.

Parameters

- **data** – A 2D numpy array of shape (N, M).
- **row_labels** – A list or array of length N with the labels for the rows.
- **col_labels** – A list or array of length M with the labels for the columns.
- **ax** – A *matplotlib.axes.Axes* instance to which the heatmap is plotted. If not provided, use current axes or create a new one. Optional.
- **cbar_kw** – A dictionary with arguments to *matplotlib.Figure.colorbar*. Optional.
- **cbarlabel** – The label for the colorbar. Optional.
- ****kwargs** – All other arguments are forwarded to *imshow*.

heatmap2d(*arr*, *x_label*, *y_label*)

class MaxPeakShavingLimit(*energy*, *power*, *efficiency*, *time_step*, *random_profile*, *file_path=''*, *scaling_factor=1*)

Bases: `object`

Determines minimum peak shaving limit that can be served for a given load profile based on a energy storage bucket model.

calc_max_peak(*max_runs*)

add_month_to(*date*)

Generates a new datetime object from given datetime object with a month added. If month is last month of year, it returns first month of next year.

Parameters **date** – given datetime object

Returns new datetime object

Return type datetime

add_year_to (*date*)
Generates a new datetime object from given datetime object with a year added.

Parameters **date** – given datetime object

Returns new datetime object

Return type datetime

all_non_abstract_subclasses_of (*cls*, *exclude=[]*)
Generates a list of non-abstract subclass from given class

Parameters

- **exclude** – list of classes which should be excluded from result list
- **cls** – given class

Returns list with non-abstract subclasses

Return type list

check (*value*)

convert_path_codec (*path*, *encode='latin-1'*, *decode='utf-8'*)
Enconding and decoding path with given codecs, e.g. in order to convert German Umlaut

Parameters

- **path** – string which should be converted
- **encode** – enconding codec, defaults: latin-1
- **decode** – decoding codec, defaults: utf-8

Returns decoded path

Return type str

copy_all_files (*source*, *target*)
Function to copy all files in a new folder

Parameters

- **source** (*path (string) to the source folder*) –
- **target** (*path (string) to the target folder*) –

create_directory_for (*path*, *warn=False*)
Function to create a folder at a specific path

Parameters **path** (*str*) –

download_file (*url_path*, *file_name*)
Downloading a file from the given url and stores it to file_name

Raises FileNotFoundError if file at URL is not available

Parameters

- **`url_path`** – URL to file to download
- **`file_name`** – path where file should be stored

`format_float`(*value*, *decimals=2*)

Formatting value to string with given decimals

Parameters

- **`value`** – given value
- **`decimals`** – round to number of decimals

Returns value as string

Return type str

`get_average_from`(*values*)

Calculates average from given value list

Parameters **`values`** – list of values

Returns average from values

Return type float

`get_maximum_from`(*values*)

Gets maximum value from given values

Parameters **`values`** – list of values

Returns maximum value from given list

Return type float

`get_path_for`(*filename*, *max_depth=4*)

Searching path for filename starting with current working directory and all of its subdirectories. If file was not found, it goes up the parent directories of the CWD with a given maxium depth

Parameters

- **`filename`** – search filename
- **`max_depth`** – maximum depth of parent directories in which should be searched, default: 3

Returns path where filename is located or a FileNotFoundError

Return type str

`get_year_from`(*tstmp*)

Calculates year from given timestamp

Parameters **`tstmp`** – given timestamp in epoch time

Returns year of given epoch timestamp

Return type float

is_codec (*string*, *codec='utf-8'*)
Checks if given string is encoded as given codec

Parameters

- **string** – string to check codec
- **codec** – codec which should be checked, default: utf-8

Returns True if string is encoded with codec, False otherwise

Return type bool

remove_all_files_from (*directory*)
Function to remove all files from a directory

Parameters **directory** (*folder path*) –

remove_file (*file*)
Removes file

Parameters **file** – path to file

search_path_for_file_in (*directory*, *filename*)
Searching path where file with filename is located within all subdirectories of directory

Parameters

- **directory** – base directory for search
- **filename** – searched filename

Returns path where filename is located, if filename was found - empty string otherwise

Return type str

write_to_file (*filename*, *_list*, *append=False*)
Writes given list of strings to file

Parameters

- **filename** – path to file
- **_list** – list with values as strings
- **append** – flag to append to file or (over)write file

Returns None

Return type None

Module contents

Submodules

class ConsolePrinter(*queue*)
Bases: `threading.Thread`

ConsolePrinter is a thread providing information of the current status of all concurrent simulations.

CUT_OFF_LENGTH: int = 120

FILE: str = 'C:\\Users\\kucevic\\Documents\\Python\\opensimses\\simse

REGISTER_SIGNAL: str = 'REGISTER'

SPACES: int = 3

STOP_SIGNAL: str = 'STOP'

UPDATE: float = 1

static clear_screen()

cut_off(*line*)

property line

property max_key_length
returns: maximal key length of all processes :rtype: int

property not_started_yet

register(*name*)
Register process with name

Parameters name – process name

run()

set(*name*, *line*)
Setting value for process name if process is registered

Parameters

- **name** – process name
- **line** – value to display for process

stop(*name*)
Signals that ConsolePrinter is not used for process with 'name' anymore

Parameters name – name of process

stop_immediately()
Stops ConsolePrinter immediately for all (!) processes

update()

class Hydrogen

Bases: `object`

BOLTZ_CONST = 1.38e-23

DENSITY_WATER = 1000

EPS = 2.220446049250313e-16

FARADAY_CONST: float = 96485.3321

HEAT_CAPACITY_HYDROGEN = 14304

HEAT_CAPACITY_OXYGEN = 920

HEAT_CAPACITY_WATER: float = 4184

IDEAL_GAS_CONST: float = 8.314462

ISENTROP_EXPONENT = 1.4098

LOWER_HEATING_VALUE = 33.327

MOLAR_MASS_HYDROGEN = 0.0010079400000000001

MOLAR_MASS_OXYGEN = 0.015999

MOLAR_MASS_WATER: float = 0.018015

NORM_QUBIC_M = 11.1235

REAL_GAS_FACTOR = 1.0006

VAN_D_WAALS_COEF_A = 0.02452065

VAN_D_WAALS_COEF_B = 2.65e-05

static isentropic_exponent(*p_1*, *p_2*)

calculates mean isentropic exponent for hydrogen for compression from pressure p_1 to pressure p_2 in bar based on diagram in Thermodynamic of Pressurized Gas Strorage in Hydrogen Science and Engineering (2016) valid for T = 50 °C

static realgas_factor(*p_1*, *p_2*)

calculates real gas factor for hydrogen based on grafig out of "Wasserstoff in der Fahrzeugtechnik" p. 49 for T = 300 K

exception ComplexNumberError(**args*)

Bases: `Exception`

exception EndOfFileError(**args*)

Bases: `Exception`

exception EndOfLifeError(**args*)

Bases: `Exception`

exception InfiniteNumberError(**args*)

Bases: `Exception`

exception NanNumberError(**args*)

Bases: `Exception`

class Logger(*name*, *log_level=0*)

Bases: `object`

SimSES Logger with possibility to write logs to console and file. Logger is configurable via logger.ini in config.

close()

debug(*msg*)

error(*msg*)

info(*msg*)

warn(*msg*)

Module contents

2.2 Simulation

2.2.1 simses.simulation package

Subpackages

simses.simulation.case_studies package

Subpackages

simses.simulation.case_studies.configs package

Module contents

Submodules

class PeakShavingCaseStudyBatchProcessing

Bases: *simses.simulation.batch_processing.BatchProcessing*

clean_up()

class FCRCaseStudyBatchProcessing

Bases: *simses.simulation.batch_processing.BatchProcessing*

clean_up()

Module contents

simses.simulation.system_tests package

Subpackages

simses.simulation.system_tests.configs package

Module contents

Submodules

class SystemTest

Bases: *simses.simulation.batch_processing.BatchProcessing*

TestBatchProcessing execute system tests for various system configurations.

clean_up()

Module contents

Submodules

class BatchProcessing(*path='C:\\Users\\kucevic\\Documents\\Python\\opensimses\\simses\\doc'*, *do_simulation=True*, *do_analysis=True*)

Bases: `abc.ABC`

BatchProcessing delivers the necessary handling for running multiple parallel SimSES simulations and distributes the configuration to each process. It supervises all processes and starts as many processes as there are cores available. If you run more simulations than cores available, it waits until a simulation has finished and fills the gap with new simulation processes.

UPDATE: int = 1

abstract clean_up()

get_result_paths()

run()

class BatchSimulation(*config_set*, *printer_queue*, *path*, *batch_size=1*, *do_simulation=True*, *do_analysis=True*, *analysis_config=<configparser.ConfigParser object>*)

Bases: `multiprocessing.context.Process`

BatchSimulation wraps the SimSES simulation into a multiprocessing Process in order to allow to run simulation in parallel. In addition, it is possible to run multiple threaded simulations within this process but this is strongly not recommended.

BATCH_DIR: str = 'batch/'

run()

class ExampleBatchProcessing
Bases: *simses.simulation.batch_processing.BatchProcessing*

This is just a simple example on how to use BatchProcessing.

clean_up()

class StorageSimulation(*path*, *config*, *printer_queue*)
Bases: `object`

StorageSimulation constructs the the storage systems and energy management system in order to execute the simulation. In the run() method the timestamp for the simulation is advanced as configured. Alternatively, simulation is included in another framework advancing timestamps itself, e.g. run_one_step() or evaluate_multiple_steps(). StorageSimulation also provided information of the current status of the simulation to the user.

property ac_system_states
returns: list of current states for all AC systems :rtype: SystemState

close()
Closing all resources of simulation

evaluate_multiple_steps(*start*, *timestep*, *power*)
Runs multiple steps of the simulation with the given start time, timestep and power list. If no power list is provided, the simulation will not be advanced.

Parameters

- **start** – start time in s
- **timestep** – timestep in s
- **power** – list of power for each timestep in W

Returns Returns a list of system states for each timestep

Return type list

reset_profiles(*ts_adapted*)
Enables looping of the simulation beyond the original length of the time series for the AmbientThermalModel and SolarIrradiationModel

run()
Executes simulation

run_one_step(*ts*, *ts_adapted=0*, *power=None*, *power_dist=None*)
Advances simulation for one step. Results can be obtained via state property.

Parameters

- **ts** – next timestamp in s
- **ts_adapted** – timestamp adaption for looping simulations multiple times (should only be used with stand alone SimSES)

- **power** – next power transfered to storage system in W, if None power is taken from configured energy management
- **power_dist** – next power distribution in W for every AC system as a list. If None, power is taken from power distributor classes.

property state
returns: current state of top level system :rtype: SystemState

Module contents

2.3 System

2.3.1 simses.system package

Subpackages

simses.system.auxiliary package

Subpackages

simses.system.auxiliary.compression package

Submodules

class Compressor
Bases: *simses.system.auxiliary.auxiliary.Auxiliary*

Compressor is an auxiliary. It calculates the necessary electric power in W which is needed to compress an ideal gas from pressure 1 to pressure 2

ac_operation_losses()
Calculates the ac operation losses

:param : :type : returns :param ——-: :param float: Power in W

abstract calculate_compression_power(*hydrogen_flow_out, pressure_1, pressure_2, temperature*)
Calculates the compression power

abstract get_compression_power()
Gets compressor power in W

class HydrogenIsentropCompressor(*compressor_eta=0.95*)
Bases: *simses.system.auxiliary.compression.compressor.Compressor*

calculate_compression_power(*hydrogen_flow_out, pressure_1, pressure_2, temperature*)

get_compression_power()

Module contents

simses.system.auxiliary.gas_drying package

Submodules

class GasDrying

Bases: *simses.system.auxiliary.auxiliary.Auxiliary*

GasDrying is an auxiliary. It calculates the energy that is needed for drying the product gas according to the specified drying level

ac_operation_losses()

Calculates the ac operation losses :return:

float: Power in W

abstract calculate_gas_drying_power(*pressure_cathode*, *h2_outflow*)

abstract get_gas_drying_power()

class HydrogenGasDrying

Bases: *simses.system.auxiliary.gas_drying.gas_dryer.GasDrying*

calculate_gas_drying_power(*pressure_cathode*, *h2_outflow*)

get_gas_drying_power()

Module contents

simses.system.auxiliary.heating_ventilation_air_conditioning package

Submodules

class FixCOPHeatingVentilationAirConditioning(*hvac_configuration*)

Bases: *simses.system.auxiliary.heating_ventilation_air_conditioning.hvac.HeatingVentilationAirConditioning*

get_electric_power()

get_max_thermal_power()

get_scop()

get_seer()

get_set_point_temperature()

get_thermal_power()

run_air_conditioning(*temperature_time_series*, *temperature_timestep*, *ambient_air_temperature*)

set_electric_power(*electric_power*)
Used in cases where multiple runs of the HVAC take place within a SimSES timestep

update_air_parameters(*air_mass=None*, *air_specific_heat=None*, *air_density=None*)

class FixCOPHeatingVentilationAirConditioningPIDControl(*hvac_configuration*)
Bases: `simses.system.auxiliary.heating_ventilation_air_conditioning.hvac.HeatingVentilationAirConditioning`

get_coefficients()

get_electric_power()

get_max_thermal_power()

get_plotting_arrays()

get_scop()

get_seer()

get_set_point_temperature()

get_thermal_power()

run_air_conditioning(*temperature_time_series*, *temperature_timestep*, *ambient_air_temperature*)

set_electric_power(*electric_power*)
Used in cases where multiple runs of the HVAC take place within a SimSES timestep

update_air_parameters(*air_mass=None*, *air_specific_heat=None*, *air_density=None*)

class HeatingVentilationAirConditioning
Bases: `simses.system.auxiliary.auxiliary.Auxiliary`, `abc.ABC`

ac_operation_losses()

get_coefficients()
Needed for plotting BA Hörmann

abstract get_electric_power()

abstract get_max_thermal_power()

get_plotting_arrays()
Needed for plotting BA Hörmann

abstract get_scop()

abstract get_seer()

abstract get_set_point_temperature()

abstract get_thermal_power()

abstract run_air_conditioning(*temperature_time_series*, *temperature_timestep*, *ambient_air_temperature*)

abstract set_electric_power(*electric_power*)

abstract update_air_parameters(*air_mass=None*, *air_specific_heat=None*, *air_density=None*)

class NoHeatingVentilationAirConditioning

Bases: *simses.system.auxiliary.heating_ventilation_air_conditioning.hvac.HeatingVentilationAirConditioning*

get_cop()

get_electric_power()

get_max_thermal_power()

get_scop()

get_seer()

get_set_point_temperature()

get_thermal_power()

run_air_conditioning(*temperature_time_series*, *temperature_timestep*, *ambient_air_temperature*)

set_electric_power(*electric_power*)

Used in cases where multiple runs of the HVAC take place within a SimSES timestep

update_air_parameters(*air_mass=None*, *air_specific_heat=None*, *air_density=None*)

Module contents

simses.system.auxiliary.pump package

Submodules

class Pump

Bases: *simses.system.auxiliary.auxiliary.Auxiliary*, abc.ABC

Pump is an auxiliary. It calculates the necessary pump power in W depending on the pressure losses.

ac_operation_losses()

Calculates the ac operation losses.

Returns Power in W.

Return type float

abstract calculate_pump_power (*pressure_loss*)
Calculates the pump power in W.

Parameters **pressure_loss** (*float*) – Current pressure loss in W. It corresponds to the volume flow times the pressure drop.

abstract close ()
Closing all resources in Pump.

abstract get_pump_power ()
Gets pump power in W.

Returns Pump power in W.

Return type float

abstract set_eta_pump (*flow_rate*, *flow_rate_max*, *flow_rate_min*)
Sets pump efficiency.

Parameters

- **flow_rate** (*float*) – Current flow rate.
- **flow_rate_max** (*float*) – Maximal needed flow rate of the hydraulic system.
- **flow_rate_min** (*float*) – Minimal needed flow rate of the hydraulic system.

class FixEtaCentrifugalPump (*efficiency*)
Bases: `simses.system.auxiliary.pump.abstract_pump.Pump`

FixEtaCentrifugalPump is a pump with a constant efficiency.

calculate_pump_power (*pressure_loss*)

close ()

get_pump_power ()

set_eta_pump (*flow_rate*, *flow_rate_max*, *flow_rate_min*)

class ScalableVariableEtaCentrifugalPump
Bases: `simses.system.auxiliary.pump.abstract_pump.Pump`

ScalableVariableEtaCentrifugalPump is a pump with an efficiency dependent on the flow rate. The pump is scaled based on the maximal and minimal expected flow rate in the system.

calculate_pump_power (*pressure_loss*)

close ()

get_pump_power ()

set_eta_pump (*flow_rate*, *flow_rate_max*, *flow_rate_min*)

class VariableEtaCentrifugalPump(*auxiliary_data_config*)

Bases: *simses.system.auxiliary.pump.abstract_pump.Pump*

VariableEtaCentrifugalPump is a pump with an efficiency dependent on the flow rate.

calculate_pump_power(*pressure_loss*)

close()

get_pump_power()

set_eta_pump(*flow_rate*, *flow_rate_max*, *flow_rate_min*)

Module contents

simses.system.auxiliary.water_heating package

Submodules

class WaterHeating

Bases: *simses.system.auxiliary.auxiliary.Auxiliary*

ac_operation_losses()

calculate_heating_power(*water_flow*, *delta_temperature*)

get_heating_power()

Module contents

Submodules

class Auxiliary

Bases: abc.ABC

abstract ac_operation_losses()

Calculates the ac operation losses

close()

Closing all open resources in operation losses

update(*time*, *state*)

Updating auxiliary losses of state

Parameters **state** –

Module contents

simses.system.dc_coupling package

Subpackages

simses.system.dc_coupling.generation package

Submodules

class DcGeneration

Bases: abc.ABC

abstract calculate_power (*time*)

Calculates power for next timestep

Parameters time – timestamp in s

abstract close ()

abstract get_auxiliaries ()

abstract get_power ()

Returns DC load power in W

Return type float

class NoDcGeneration

Bases: *simses.system.dc_coupling.generation.dc_generation.DcGeneration*

calculate_power (*time*)

close ()

get_auxiliaries ()

get_power ()

class PvDcGeneration (*peak_power*)

Bases: *simses.system.dc_coupling.generation.dc_generation.DcGeneration*

PvDcGeneration provides a basic algorithm imitating a pv plant with the same power each day.

calculate_power (*time*)

close ()

get_auxiliaries ()

get_power ()

Module contents

simses.system.dc_coupling.load package

Submodules

class DcBusChargingFixed(*charging_power*)

Bases: *simses.system.dc_coupling.load.dc_load.DcLoad*

DcBusChargingFixed acts as fixed load profile for bus connecting to the storage system at the same time each day.

calculate_power(*time*)

close()

get_auxiliaries()

get_power()

class DcBusChargingProfile(*config*, *capacity*, *file_name*)

Bases: *simses.system.dc_coupling.load.dc_load.DcLoad*

DcBusChargingProfile is able to read in SOC profiles and calculates a load power for charging.

calculate_power(*time*)

close()

get_auxiliaries()

get_power()

class DcBusChargingRandom(*charging_power*)

Bases: *simses.system.dc_coupling.load.dc_load.DcLoad*

DcBusChargingRandom acts as load profile for busses connected to the storage system at varying start and end times of each day.

calculate_power(*time*)

close()

get_auxiliaries()

get_power()

class DcLoad

Bases: abc.ABC

Abstract class for every DC load

abstract calculate_power(*time*)

Calculates power for next timestep

Parameters time – timestamp in s

abstract close()

abstract get_auxiliaries()

abstract get_power()

Returns DC load power in W

Return type float

class DCRadioUPSLoad(*charging_power*)

Bases: *simses.system.dc_coupling.load.dc_load.DcLoad*

DcBusChargingFixed acts as fixed load profile for bus connecting to the storage system at the same time each day.

calculate_power(*time*)

close()

get_auxiliaries()

get_power()

class NoDcLoad

Bases: *simses.system.dc_coupling.load.dc_load.DcLoad*

calculate_power(*time*)

close()

get_auxiliaries()

get_power()

Module contents

Submodules

class BusChargingDcCoupling(*charging_power, generation_power*)

Bases: *simses.system.dc_coupling.dc_coupler.DcCoupling*

class BusChargingProfileDcCoupling(*config, capacity, file_name*)

Bases: *simses.system.dc_coupling.dc_coupler.DcCoupling*

class DcCoupling(*dc_load, dc_generation*)

Bases: `object`

DcCouplings provide a possiblity to add a DC load or DC generation to the intermediate circuit of an AC storage system.

close()

get_auxiliaries()

get_power()

Returns net power of DC load and DC generation in W

Return type float

update (*time*, *state*)

In-place update of system state for dc power additional

Parameters

- **time** – current timestamp in s
- **state** – current system state

class NoDcCoupling

Bases: *simses.system.dc_coupling.dc_coupler.DcCoupling*

class USPDCCoupling (*charging_power*, *generation_power*)

Bases: *simses.system.dc_coupling.dc_coupler.DcCoupling*

Module contents

simses.system.housing package

Submodules

class Housing (*layer_inner=None*, *layer_mid=None*, *layer_outer=None*, *default_scale=0*)

Bases: `abc.ABC`

class to specify the housing of the storage system

add_component_volume (*volume*)

abstract property albedo

abstract property azimuth

close ()

get_number_of_containers ()

property inner_layer

Access to layer attributes of inner-layer such as temperature, dimensions, mass, thermal properties

property internal_air_volume

Returns internal air volume of housing in m3

property internal_volume

Returns internal volume of housing in m3

property mid_layer

Access to layer attributes of mid-layer such as temperature, dimensions, mass, thermal properties

property outer_layer

Access to layer attributes of outer-layer such as temperature, dimensions, mass, thermal properties

class FortyFtContainer(*housing_configuration*, *temperature=None*)

Bases: `simses.system.housing.abstract_housing.Housing`

property albedo

property azimuth

class Layer(*layer_attributes*)

Bases: `object`

Class Layer specifies and controls the attributes of each layer of the wall of a shipping container (Housing)

ABSORPTIVITY: str = 'absorptivity'

BREADTH: str = 'breadth'

CONVECTION_COEFFICIENT: str = 'convection_coefficient_air'

DENSITY: str = 'density'

HEIGHT: str = 'height'

LENGTH: str = 'length'

SPECIFIC_HEAT: str = 'specific_heat'

TEMPERATURE: str = 'temperature'

THERMAL_CONDUCTIVITY: str = 'thermal_conductivity'

THICKNESS: str = 'thickness'

property absorptivity

property breadth

property convection_coefficient_air

property height

property length

property mass

property specific_heat

property surface_area_long_side

property surface_area_roof

property surface_area_short_side

property surface_area_total

property temperature

property thermal_conductivity

property thickness

update_scale()

class NoHousing

Bases: *simses.system.housing.abstract_housing.Housing*

LARGE_NUMBER = 1.7976931348623157e+208

property albedo

property azimuth

property internal_volume

class TwentyFtContainer(*housing_configuration*, *temperature=None*)

Bases: *simses.system.housing.abstract_housing.Housing*

property albedo

property azimuth

Module contents

simses.system.power_electronics package

Subpackages

simses.system.power_electronics.acdc_converter package

Submodules

class AcDcConverter(*max_power*)

Bases: `abc.ABC`

abstract close()

Closing all open resources in acdc converter

abstract classmethod create_instance(*max_power*, *power_electronics_config*)

abstract property mass

Mass of acdc converter in kg

property max_power

returns the maximum ac power of the acdc converter

Returns maximum power of the acdc converter (ac power)

Return type float

property standby_power_threshold

returns the ac power threshold for standby

abstract property surface_area
Surface area of acdc converter in m2

abstract to_ac(*power*, *voltage*)
Calculated dc power for discharging process

Parameters

- **power** (*float*) – requested ac power in W
- **voltage** (*float*) – dc voltage of intermediate circuit in V

Returns dc power in W

Return type float

abstract to_ac_reverse(*dc_power*, *voltage*)
recalculates ac power in W, if the BMS limits the DC_power

Parameters

- **voltage** – voltage of intermediate circuit in V
- **dc_power** – dc power of intermediate circuit in W

Returns ac power in W

Return type float

abstract to_dc(*power*, *voltage*)
Calculated dc power for charging process

Parameters

- **power** (*float*) – requested ac power in W
- **voltage** (*float*) – dc voltage of intermediate circuit in V

Returns dc power in W

Return type float

abstract to_dc_reverse(*dc_power*, *voltage*)
recalculates ac power in W, if the BMS limits the DC_power

Parameters

- **voltage** – voltage of intermediate circuit in V
- **dc_power** – dc power of intermediate circuit in W

Returns ac power in W

Return type float

abstract property volume
Volume of acdc converter in m3

class BonfiglioliAcDcConverter(*max_power*)
Bases: *simses.system.power_electronics.acdc_converter.abstract_acdc_converter.AcDcConverter*

Efficiency fit for converter RPS TL-4Q by Bonfiglioli (http://www.docsbonfiglioli.com/pdf_documents/catalogue/VE_CAT_RTL-4Q_STD_ENG-ITA_R00_5_WEB.pdf)

using field data and Notton-Fit as described in master's thesis by Felix Müller.

close()

classmethod create_instance(*max_power*, *power_electronics_config=None*)

property mass

property surface_area

to_ac(*power*, *voltage*)

to_ac_reverse(*dc_power*, *voltage*)

to_dc(*power*, *voltage*)

to_dc_reverse(*dc_power*, *voltage*)

property volume

class FixEfficiencyAcDcConverter(*max_power*, *efficiency=0.95*)

Bases: *simses.system.power_electronics.acdc_converter.abstract_acdc_converter.AcDcConverter*

close()

classmethod create_instance(*max_power*, *power_electronics_config=None*)

property mass

property surface_area

to_ac(*power*, *voltage*)

to_ac_reverse(*dc_power*, *voltage*)

to_dc(*power*, *voltage*)

to_dc_reverse(*dc_power*, *voltage*)

property volume

class NoLossAcDcConverter(*max_power*)

Bases: *simses.system.power_electronics.acdc_converter.fix_efficiency.FixEfficiencyAcDcConverter*

classmethod create_instance(*max_power*, *power_electronics_config=None*)

class NottonAcDcConverter(*max_power*)

Bases: *simses.system.power_electronics.acdc_converter.abstract_acdc_converter.AcDcConverter*

Notton, G.; Lazarov, V.; Stoyanov, L. (2010): Optimal sizing of a grid-connected PV system for various PV module technologies and inclinations, inverter efficiency characteristics and locations. In: Renewable Energy 35 (2), S. 541-554. DOI: 10.1016/j.renene.2009.07.013.

close()

classmethod create_instance(*max_power*, *power_electronics_config=None*)

property mass

property surface_area

to_ac(*power*, *voltage*)

to_ac_reverse(*dc_power*, *voltage*)

to_dc(*power*, *voltage*)

to_dc_reverse(*dc_power*, *voltage*)

property volume

class NottonLossAcDcConverter(*max_power*)

Bases: *simses.system.power_electronics.acdc_converter.abstract_acdc_converter.AcDcConverter*

close()

classmethod create_instance(*max_power*, *power_electronics_config=None*)

property mass

property surface_area

to_ac(*power*, *voltage*)

to_ac_reverse(*dc_power*, *voltage*)

to_dc(*power*, *voltage*)

to_dc_reverse(*dc_power*, *voltage*)

property volume

class Sinamics120AcDcConverter(*max_power*, *config*)

Bases: *simses.system.power_electronics.acdc_converter.abstract_acdc_converter.AcDcConverter*

This class uses a pre-generated Look-up table (LUT) for the efficiency of the Sinamics S120 converter in charging and discharging processes. The LUT was generated by Michael Schimpe from measured data using a script written in MATLAB. The efficiency is measured for 650 V. Source: https://doi.org/10.1016/j.egypro.2018.11.065

close()

classmethod create_instance(*max_power*, *power_electronics_config*)

property mass

property surface_area

to_ac (*power*, *voltage*)

to_ac_reverse (*dc_power*, *voltage*)

to_dc (*power*, *voltage*)

to_dc_reverse (*dc_power*, *voltage*)

property volume

class AcDcConverterIdenticalStacked (*number_converters*, *switch_value*, *acdc_converter*, *power_electronics_config=None*)

Bases: `simses.system.power_electronics.acdc_converter.abstract_acdc_converter.AcDcConverter`

close ()

classmethod create_instance (*max_power*, *power_electronics_config=None*)

property mass

property surface_area

to_ac (*power*, *voltage*)

to_ac_reverse (*dc_power*, *voltage*)

to_dc (*power*, *voltage*)

to_dc_reverse (*dc_power*, *voltage*)

property volume

class SungrowAcDcConverter (*max_power*)

Bases: `simses.system.power_electronics.acdc_converter.abstract_acdc_converter.AcDcConverter`

Fitted Efficiency Curve using field data: Master Thesis by Felix Müller

close ()

classmethod create_instance (*max_power*, *power_electronics_config=None*)

property mass

property surface_area

to_ac (*power*, *voltage*)

to_ac_reverse (*dc_power*, *voltage*)

to_dc (*power*, *voltage*)

to_dc_reverse (*dc_power*, *voltage*)

property volume

Module contents

simses.system.power_electronics.dcdc_converter package

Submodules

class DcDcConverter(*intermediate_circuit_voltage*)

Bases: `abc.ABC`

abstract calculate_dc_current(*dc_power*, *storage_voltage*)

function to calculate the dc current

Parameters

- **dc_power** (*dc input power in W*) –
- **storage_voltage** (*voltage of storage in V*) –

abstract close()

Closing all open resources in dcdc converter

abstract property dc_current

abstract property dc_power

abstract property dc_power_loss

abstract get_auxiliaries()

property intermediate_circuit_voltage

abstract property mass

Mass of dc dc converter in kg

abstract property max_power

abstract reverse_calculate_dc_current(*dc_power*, *storage_voltage*)

function to calculate the dc current

Parameters

- **dc_power** – dc input power in W
- **storage_voltage** – voltage of storage in V

abstract property surface_area

Surface area of dc dc converter in m2

abstract property volume

Volume of dc dc converter in m3

class FixEfficiencyDcDcConverter(*intermediate_circuit_voltage*, *efficiency=0.98*)

Bases: *simses.system.power_electronics.dcdc_converter.abstract_dcdc_converter.DcDcConverter*

calculate_dc_current (*dc_power_intermediate_circuit, storage_voltage*)

close ()

property dc_current

property dc_power

property dc_power_loss

get_auxiliaries ()

property mass

property max_power

reverse_calculate_dc_current (*dc_power_storage, storage_voltage*)

property surface_area

property volume

class NoLossDcDcConverter (*intermediate_circuit_voltage*)

Bases: *simses.system.power_electronics.dcdc_converter.abstract_dcdc_converter.DcDcConverter*

calculate_dc_current (*dc_power, storage_voltage*)

close ()

property dc_current

property dc_power

property dc_power_loss

get_auxiliaries ()

property mass

property max_power

reverse_calculate_dc_current (*dc_power, storage_voltage*)

property surface_area

property volume

Module contents

Submodules

class PowerElectronics (*acdc_converter*)

Bases: object

class to update the power elections values

property acdc_converter_type

Returns the name of the selected acdc converter

Returns name of the selected acdc converter

Return type str

close()
Closing all open resources in the power electronics unit

get_auxiliaries()

get_dc_power_from(*ac_power*, *voltage*)
function to get the dc power from the power electronics unit

Parameters

- **ac_power** – requested ac power from the EMS
- **voltage** – DC voltage

Returns dc power

Return type float

property max_power

update(*time*, *state*)
Function to update states regarding the power electronics unit

Parameters **state** – current SystemState

update_ac_power_from(*state*)
function to update ac power if fulfillment is not 100 %

Parameters **state** – current SystemState

property volume
Volume of power electronics in m3

Module contents

simses.system.test package

Submodules

class TestClassAcDc
Bases: `object`

abs_voltage = 20

converter_subclass_list()

make_converter(*converter_subclass*, *max_power*)

max_power = 40000

step = 10

test_max_power_to_ac(*converter_subclass_list*)

test_max_power_to_ac_reverse(*converter_subclass_list*)

test_max_power_to_dc(*converter_subclass_list*)

test_max_power_to_dc_reverse(*converter_subclass_list*)

test_sign_to_ac(*ac_power*, *converter_subclass_list*)

test_sign_to_ac_reverse(*dc_power*, *converter_subclass_list*)

test_sign_to_dc(*ac_power*, *converter_subclass_list*)

test_sign_to_dc_reverse(*dc_power*, *converter_subclass_list*)

test_to_ac_is_nan(*converter_subclass_list*)

test_to_ac_reverse_is_nan(*converter_subclass_list*)

test_to_dc_is_nan(*converter_subclass_list*)

test_to_dc_reverse_is_nan(*converter_subclass_list*)

test_values = [0, 4000, 8000, 12000, 16000, 20000, 24000, 28000, 3200

test_get_temperature(*time_step*)

class TestFixCOPHeatingVentilationAirConditioning

Bases: `object`

air_mass = 50000

air_specific_heat = 1006

create_model()

hvac = 'constant_hvac'

hvac_configuration = ['FixCOPHeatingVentilationAirConditioning', '500

hvac_model = 'FixCOPHeatingVentilationAirConditioning'

hvac_model_config: configparser.ConfigParser = <configparser.ConfigP

max_thermal_power: int = 5000

set_point_temperature: int = 25

storage_system_config = <simses.commons.config.simulation.system.Stor

temperature_range = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1

test_run_air_conditioning(*temperature_value*)

tests_set_point_deviation = 20

test_get_pump(*uut*, *pressure_loss*, *result*)

uut(*eta_pump*)

create_data_config(*filename*)

create_file(*value*)

create_general_config ()

test_get_temperature (*time_factor*, *result*, *uut*)

uut (*ambient_temperature*)

create_data_config (*filename*)

create_file (*value*)

create_general_config ()

test_methods (*time_factor*, *housing*, *result*, *uut*)

uut (*global_horizontal_irradiance*, *housing*)

test_to_ac (*result*, *power*, *uut*)

test_to_ac_fail (*result*, *power*, *uut*)

test_to_ac_reverse (*result*, *power*, *uut*)

test_to_ac_reverse_fail (*result*, *power*, *uut*)

test_to_dc (*result*, *power*, *uut*)

test_to_dc_fail (*result*, *power*, *uut*)

test_to_dc_reverse (*result*, *power*, *uut*)

test_to_dc_reverse_fail (*result*, *power*, *uut*)

uut (*max_power*)

Module contents

simses.system.thermal package

Subpackages

simses.system.thermal.ambient package

Submodules

class AmbientThermalModel

Bases: `abc.ABC`

AmbientThermalModel provides a temperature of the ambient for the system thermal model calculations.

abstract close ()

closes all open resources in ambient thermal model

abstract create_instance ()

reinstantiates the AmbientThermalModel :return: AmbientThermalModel

abstract get_initial_temperature()
Returns the ambient temperature

Parameters **time** (*current simulation time*) –

Returns ambient temperature in Kelvin

Return type float

abstract get_temperature(*time*)
Returns the ambient temperature

Parameters **time** (*current simulation time*) –

Returns ambient temperature in Kelvin

Return type float

class ConstantAmbientTemperature(*temperature=25*)
Bases: *simses.system.thermal.ambient.ambient_thermal_model.AmbientThermalModel*

ConstantAmbientTemperature provides a constant temperature over time.

close()

create_instance()
reinstantiates the AmbientThermalModel :return: AmbientThermalModel

get_initial_temperature()

get_temperature(*time*)

class LocationAmbientTemperature(*profile_config, general_config*)
Bases: *simses.system.thermal.ambient.ambient_thermal_model.AmbientThermalModel*

LocationAmbientTemperature provides a ambient temperature profile for a specified location. Ambient temperature time series data for Berlin, Germany and Jodhpur, India from DLR Greenius Tool: https://www.dlr.de/sf/en/desktopdefault.aspx/tabid-11688/20442_read-44865/

close()

create_instance()
reinstantiates the AmbientThermalModel :return: AmbientThermalModel

get_initial_temperature()

get_temperature(*time*)

Module contents

simses.system.thermal.model package

Submodules

class NoSystemThermalModel(*ambient_thermal_model*, *general_config*)

Bases: *simses.system.thermal.model.system_thermal_model.SystemThermalModel*

This model does nothing - keeps the system air temperature equal to ambient temperature

LARGE_NUMBER = 1.7976931348623157e+208

calculate_temperature(*time*, *state*, *states*)

get_ambient_temperature()

get_auxiliaries()

get_hvac_thermal_power()

get_solar_irradiation_thermal_load()

get_temperature()

reset_profiles(*ts_adapted*)

Enables looping of the simulation beyond the original length of the time series for the AmbientThermalModel and SolarIrradiationModel

update_air_parameters()

class SystemThermalModel

Bases: `abc.ABC`

Thermal model of a storage system

abstract calculate_temperature(*time*, *storage_system_ac_state*, *storage_system_dc_states*)

Calcualtes the current temperature of the system

Parameters

- **time** (*current simulation time*) –
- **state** (*current system state*) –

Returns

- *param time:*
- *param storage_system_dc_states:*
- *param storage_system_ac_state:*

close()

Closing all open resources in system thermal model

abstract get_ambient_temperature()
Returns the ambient temperature at the location

abstract get_auxiliaries()

abstract get_hvac_thermal_power()
returns the total HVAC thermal power in W

abstract get_solar_irradiation_thermal_load()
returns the solar irradiation thermal load on the container in W

abstract get_temperature()
Returns the current temperature of the system (Internal air temperature)

Returns system temperature in Kelvin

Return type float

abstract reset_profiles(*ts_adapted*)
Enables looping of the simulation beyond the original length of the time series for the AmbientThermalModel and SolarIrradiationModel

abstract update_air_parameters()

class ZeroDDynamicSystemThermalModel(*ambient_thermal_model*, *housing*, *hvac*, *general_config*, *storage_system_config*, *dc_systems*, *acdc_converter*)
Bases: *simses.system.thermal.model.system_thermal_model.SystemThermalModel*

This is going to be Stefan's model.

calculate_temperature(*time*, *state*, *dc_states*)

calculate_thermal_capacities()

calculate_thermal_power(*temperature_series*)

calculate_thermal_resistances()

close()

get_ambient_temperature()

get_auxiliaries()

get_hvac_thermal_power()
returns the HVAC thermal power in W :return: __hvac_thermal_power as float

get_solar_irradiation_thermal_load()

get_temperature()

plot_temperatures(*state*)

reset_profiles(*ts_adapted*)
Enables looping of the simulation beyond the original length of the time series for the AmbientThermalModel and SolarIrradiationModel

update_air_parameters()

class ZeroDSystemThermalModel(*ambient_thermal_model*, *housing*, *hvac*, *general_config*, *dc_systems*, *acdc_converter*, *solar_irradiation_model*)

Bases: `simses.system.thermal.model.system_thermal_model.SystemThermalModel`

AIR_SPECIFIC_HEAT = 1006

calculate_temperature(*time*, *state*, *storage_system_dc_states*)

primary method of the system thermal model which calculates and sets temperatures of all components :param time: timestamp of current timestep as float :param state: state of the StorageSystemAC as SystemState :param storage_system_dc_states: states of the StorageSystemDC objects within StorageSystemAC as [SystemState] :return: None

close()

closes specified open resources

get_ambient_temperature()

returns the ambient temperature at the location in K :return: __ambient_air_temperature as float

get_auxiliaries()

:returns instances of the Auxiliary class (HVAC system in this case) :return: __heating_cooling as [Auxiliary]

get_hvac_thermal_power()

returns the HVAC thermal power in W :return: __hvac_thermal_power as float

get_solar_irradiation_thermal_load()

returns the thermal load due to solar irradiation in W :return: __solar_irradiation_thermal_load as float

get_temperature()

returns internal air temperature in the container in K :return: __internal_air_temperature as float

reset_profiles(*ts_adapted*)

Enables looping of the simulation beyond the original length of the time series for the AmbientThermalModel and SolarIrradiationModel

update_air_parameters()

updates values of mass and air density stored within the HVAC object :return: None

class ZeroDSystemThermalModelSingleStep(*ambient_thermal_model*, *housing*, *hvac*, *general_config*, *dc_systems*, *acdc_converter*)

Bases: `simses.system.thermal.model.system_thermal_model.SystemThermalModel`

This model functions at StorageSystemAC Level.

calculate_temperature(*time*, *state*, *states*)

close()

get_ambient_temperature()

get_auxiliaries()

get_hvac_thermal_power()

get_solar_irradiation_thermal_load()

get_temperature()

reset_profiles(*ts_adapted*)
Enables looping of the simulation beyond the original length of the time series for the AmbientThermalModel and SolarIrradiationModel

update_air_parameters()

Module contents

simses.system.thermal.solar_irradiation package

Submodules

class LocationSolarIrradiationModel(*profile_config*, *general_config*, *housing*)
Bases: `simses.system.thermal.solar_irradiation.solar_irradiation_model.SolarIrradiationModel`

LocationSolarIrradiationModel calculates the total incident solar irradiation on the selected housing object at any given time for a specified location. Solar irradiation time series data from various sources: - German Weather Service (Deutscher Wetterdienst - DWD) - Oskar-von-Miller tower - LMU Garching

close()

create_instance()
reinstantiates the SolarIrradiationModel :return: SolarIrradiationModel

get_global_horizontal_irradiance(*time_step*)

get_heat_load(*time_stamp*)

determine_leap_year(*time_struct*)

get_diffuse_radiation_horizontal(*global_radiation_horizontal*, *extraterrestrial_radiation_horizontal*, *sun_elevation*)

get_direct_radiation_horizontal(*global_radiation_horizontal*, *diffuse_radiation_horizontal*)

get_equation_of_time(*orbital_inclination*)

get_extraterrestrial_radiation_horizontal(*time_struct*, *sun_elevation*)

get_hour_angle(*true_local_time*)

get_orbital_inclination(*time_stamp*)

get_sun_declination(*orbital_inclination*)

get_total_power_radiation(*power_radiation_one*, *power_radiation_two*, *power_radiation_three*, *power_radiation_four*, *power_radiation_five*)

class NoSolarIrradiationModel

Bases: *simses.system.thermal.solar_irradiation.solar_irradiation_model.SolarIrradiationModel*

NoSolarIrradiationModel returns a value of 0.0 for the thermal load due to the solar irradiation

close()

create_instance()

reinstantiates the SolarIrradiationModel :return: SolarIrradiationModel

get_global_horizontal_irradiance(*time_step*)

get_heat_load(*time_stamp*)

class SolarIrradiationModel

Bases: `abc.ABC`

SolarIrradiationModel calculates the total incident solar irradiation on the selected housing object at any given time for a specified location. Solar irradiation time series data from various sources:

abstract close()

closes all open resources in solar irradiation model

abstract create_instance()

reinstantiates the SolarIrradiationModel :return: SolarIrradiationModel

abstract get_global_horizontal_irradiance(*time_step*)

Returns the value of global horizontal irradiance for specified timestep for selected location (in W/m2) :param time_step: :return: irradiance, or 0 (depending on chosen sub-class)

abstract get_heat_load(*time*)

This method is called from the system thermal model, and returns thermal power on container surfaces :param time: :return: thermal power, or 0 (depending on chosen sub-class)

Module contents

Module contents

Submodules

class StorageSystemFactory(*config*)

Bases: `object`

The StorageSystemFactory instantiates all necessary and configured objects for AC and DC storage systems.

close()

create_acdc_converter(*converter*, *max_power*, *intermediate_cicuit_voltage*)

create_ambient_temperature_model()

create_dc_couplings(*name*)

create_dcdc_converter(*converter*, *intermediate_circuit_voltage*)

create_housing_from(*housing_name*, *ambient_thermal_model*)

create_hvac_from(*hvac*, *housing*)

create_power_distributor_ac()

create_power_distributor_dc(*dc_systems*, *power_electronics*)

create_solar_irradiation_model(*housing*)

create_stacked_acdc_converter(*converter*, *acdc_converter*)

create_storage_systems_ac(*data_export*)

create_storage_systems_dc(*name*, *data_export*, *hvac_set_point*, *ambient_thermal_model*, *intermediate_circuit_voltage*, *system_id*)

create_storage_technology(*technologies*, *data_export*, *hvac_set_point*, *ambient_thermal_model*, *storage_id*, *system_id*, *voltage*)

create_system_state_from(*system_id*, *storage_id*)

create_thermal_model_from(*hvac*, *ambient_thermal_model*, *housing*, *dc_systems*, *ac_dc_converter*, *solar_irradiation_model*)

class StorageCircuit(*data_export*, *config*)

Bases: `object`

StorageCircuit is the top level class including all AC storage systems. The is distributed via a PowerDistributor logic to each AC storage system.

property ac_system_states

close()
Closing all resources in storage systems

get_system_parameters()

reset_profiles(*ts_adapted*)
Enables looping of the simulation beyond the original length of the time series for the AmbientThermalModel and SolarIrradiationModel

property state

update(*time*, *power*, *power_dist=None*)

class StorageSystemAC(*state*, *data_export*, *system_thermal_model*, *power_electronics*, *storage_systems*, *dc_couplings*, *housing*, *power_distributor*)
Bases: `object`

AC storage system class manages all connections within the AC storage system. Connections are power electronics, further DC couplings (load/generation), housing, thermal model, auxiliaries, DC storage systems and its power distribution logic. DC couplings are handled as an interference and tried to be primarily fed by storage system power.

close()
Closing all open resources in AC storage system

get_system_parameters()

property max_power

reset_profiles(*ts_adapted*)
Enables looping of the simulation beyond the original length of the time series for the AmbientThermalModel and SolarIrradiationModel

property state
Returns current system state

Returns State of the ac system

Return type *SystemState*

property system_id

update(*power*, *time*)
Function to update the states of an AC storage system

Parameters

- **power** (*ac target power (from energy management system)*) –
- **time** (*current simulation time*) –

class StorageSystemDC(*system_id*, *storage_id*, *data_export*, *dcdc_converter*, *storage_technology*)
Bases: `object`

DC storage system class incorporates a DCDC converter and a storage technology. In this class the power is split into current voltage via the DCDC converter based on the current storage technology state.

close()
Closing all open resources in a DC storage system

get_auxiliaries()

get_dc_dc_converter()

get_storage_technology()

get_system_parameters()

property state
Function to write dc states into SystemState

Returns SystemState with all values from a dc system

Return type *SystemState*

update(*time*, *dc_power*)
Function to update the states of a DC storage system

Parameters

- **time** (*current simulation time*) –
- **dc_power** (*DC target power (from ac storage system)*) –

property volume
Volume of dc system in m3

wait()
In case of multihreading in storage technologies, the upper storage system needs to wait for results.

Module contents

2.4 Logic

2.4.1 simses.logic package

Subpackages

simses.logic.energy_management package

Subpackages

simses.logic.energy_management.strategy package

Subpackages

simses.logic.energy_management.strategy.basic package

Submodules

class FrequencyContainmentReserve(*config*, *config_ems*, *profile_config*)

Bases: simses.logic.energy_management.strategy.operation_strategy.OperationStrategy

Operation strategy for providing FCR. It was developed according to the German regulatory framework . The requested charging and discharging power is proportional to the frequency deviation. Below 49.8 Hz or above 50.2 Hz the output power is set to the prequalified power. Within the frequency dead band around 50 Hz with +/-10 mHz the output power is set to 0 W. The degree of freedom to exceed the output power by a factor of 1.2 is used aiming to bring the SOC back to a predefined SOC set point.

clear()

close()

next(*time*, *system_state*, *power=0*)

update(*energy_management_state*)

class IntradayMarketRecharge(*general_config*, *fcr_config*)

Bases: simses.logic.energy_management.strategy.operation_strategy.OperationStrategy

If the SOC falls below a predefined lower limit or it exceeds an upper limit the gls{bess} charges or discharges by trading energy on the electricity market, in particular the IDM.

clear()

close()

next(*time*, *system_state*, *power=0*)

update(*energy_management_state*)

class PeakEnergyForecaster(*profile*, *forecasting_steps*, *ps_limit*, *p_max*, *efficiency*, *general_config*, *profile_config*)

Bases: object

static limit(*delta*, *max_abs*)

next(*time*, *current_load*)

class PeakShavingPerfectForesight(*general_config*, *power_profile*, *forecast_profile*, *energy_management_config*, *system_config*, *profile_config*)

Bases: *simses.logic.energy_management.strategy.operation_strategy.OperationStrategy*

Peak Shaving under the assumption of perfect foresight for the load profile in order to reduce calendar degradation. The BESS will only charge up to the energy that is needed for the next load peak, right before the load peak occurs.

clear ()

close ()

next (*time*, *system_state*, *power=0*)

update (*energy_management_state*)

class SimplePeakShaving (*power_profile*, *ems_config*)

Bases: *simses.logic.energy_management.strategy.operation_strategy.OperationStrategy*

Basic Peak Shaving operation strategy: If the storage is almost full (> xy %), the storage is not charged anymore to avoid a misrepresent fulfillmentfactor

clear ()

close ()

next (*time*, *system_state*, *power_offset=0*)

update (*energy_management_state*)

class PowerFollower (*power_profile*)

Bases: *simses.logic.energy_management.strategy.operation_strategy.OperationStrategy*

PowerFollower is a basic operation strategy which just forwards a given power profile to the storage system.

clear ()

close ()

next (*time*, *system_state*, *power=0*)

update (*energy_management_state*)

class ResidentialPvFeedInDamp (*power_profile*, *general_config*, *pv_generation_profile*)

Bases: *simses.logic.energy_management.strategy.operation_strategy.OperationStrategy*

Operation Strategy for a residential home storage system in combination with PV generation. The algorithm plans the charging of the lithium_ion according to a PV prediction. It tries to provide a fully charged BESS at sundown.

clear ()

close ()

next (*time*, *system_state*, *power=0*)

pv_prediction(*pv_generation_profile*)
Function do determine the last pv production of every day of the simulation and storing the timestamps into a list.

Returns timestamps of the last pv production of every day of the simulation.

Return type list

update(*energy_management_state*)

class ResidentialPvGreedy(*power_profile*, *pv_generation_profile*)
Bases: *simses.logic.energy_management.strategy.operation_strategy.OperationStrategy*

Operation Strategy for a residential home storage system in combination with PV generation. The algorithm fills the BESS as fast as possible without consideration for the grid by meeting the residual load at all times.

clear()

close()

next(*time*, *system_state*, *power=0*)

update(*energy_management_state*)

class SocFollower(*config*, *profile_config*)
Bases: *simses.logic.energy_management.strategy.operation_strategy.OperationStrategy*

SOC Follower is a basic operation strategy which converts a given soc profile to a power profile and forward it to the storage system.

clear()

close()

next(*time*, *system_state*, *power=0*)

update(*energy_management_state*)

class UseAllRenewableEnergy(*pv_generation_profile*)
Bases: *simses.logic.energy_management.strategy.operation_strategy.OperationStrategy*

This operation strategy is for a plant that uses the whole energy provided by a renewable energy source. This strategy is implemented especially for an electrolyzer which produces hydrogen with the energy out of a solar or a wind energy plant.

clear()

close()

next(*time*, *system_state*, *power=0*)

update(*energy_management_state*)

Module contents

simses.logic.energy_management.strategy.serial package

Submodules

Module contents

simses.logic.energy_management.strategy.stacked package

Submodules

class FcrIdmRechargeStacked(*general_config, ems_config, profile_config*)

Bases: `simses.logic.energy_management.strategy.stacked.stacked_operation_strategy.StackedOperationStrategy`

class StackedOperationStrategy(*priority, strategies*)

Bases: `simses.logic.energy_management.strategy.operation_strategy.OperationStrategy`

Algorithm to determine output power for each operation (in multiuse case) of the ESS for the next timestep.

clear()

close()

next(*time, system_state, power=0*)

update(*energy_management_state*)

Module contents

Submodules

class OperationPriority

Bases: `object`

Priority options of a operation strategy in a multiuse case VERY_HIGH = 0 HIGH = 1 MEDIUM = 2 LOW = 3 VERY_LOW = 4

HIGH = 1

LOW = 3

MEDIUM = 2

VERY_HIGH = 0

VERY_LOW = 4

class OperationStrategy(*priority*)
Bases: abc.ABC

Algorithm to determine output power of the ESS for the next timestep.

abstract clear()
Clearing internal values of operations strateggy

abstract close()
Closing open resources

abstract next(*time*, *system_state*, *power=0*)
Provides the next value for the output power of the ESS.

Parameters

- **time** (*float*) –
- **system_state** (SystemState) –
- **power** (*float*) – Power value for stacked operation strategy.

Returns Output power for the ESS in the next timestep.

Return type float

property priority
Defines the priority of the operation strategy in a multiuse case VERY_HIGH = 0 HIGH = 1 MEDIUM = 2 LOW = 3 VERY_LOW = 4

static sort(*strategies*)
Sorts the order in which the operation strategies are applied in a multiuse case

abstract update(*energy_management_state*)
In-place update of energy management state variables

Parameters energy_management_state (*state for energy management*) –

Module contents

Submodules

class EnergyManagementFactory(*config*, *path=None*)
Bases: object

Energy Management Factory to create the operation strategy of the ESS.

binary_profile()

close()

create_energy_management_state()

create_operation_strategy ()
Energy Management Factory to create the operation strategy of the ESS based on the __analysis_config file_name.

generation_profile ()

load_profile ()
Returns the load profile for the EnergyManagementSystem

class EnergyManagement (*data_export*, *config*)
Bases: `object`

Energy Management System to control the operation of the ESS.

close ()
Closing open resources

create_instance ()

export (*time*)

next (*time*, *system_state*, *power_offset=0*)
Return the next power value of the energy management strategy in W based on the operation strategy.

Parameters

- **time** (*float*) – Current timestamp.
- **time_loop_delta** (*float*) – Time delta due to current simulation loop.
- **system_state** (`State`) – Current state of the system.
- **power** (*float*) – Power value of stacked operation strategy.

Returns Power value the StorageSystem should meet in W.

Return type float

Module contents

simses.logic.power_distribution package

Submodules

class EfficientPowerDistributor (*max_pe_power*)
Bases: *simses.logic.power_distribution.power_distributor.PowerDistributor*

EqualPowerDistributor distributes the power equally to all system independent of their current state.

get_power_for (*power*, *state*)

set (*time*, *states*, *power*)

class EqualPowerDistributor

Bases: *simses.logic.power_distribution.power_distributor.PowerDistributor*

EqualPowerDistributor distributes the power equally to all system independent of their current state.

get_power_for (*power*, *state*)

set (*time*, *states*, *power*)

class PowerDistributor

Bases: `abc.ABC`

PowerDistributor incorporates a logic on how to distribute power between several systems. The logic is based on the system state of each system.

close ()

abstract get_power_for (*power*, *state*)

Calculates the power share of the overall to be distributed power to a specific system specified with system state

Parameters

- **power** – overall power to be distributed to all systems
- **state** – system state of system to calculate a specific power share of power

Returns power share of specified system with corresponding system state

Return type float

abstract set (*time*, *states*, *power*)

Setting information from all system states necessary for the PowerDistributor

Parameters

- **time** – current simulation time as epoch time
- **states** – list of current system states
- **power** – overall power to be distributed to all systems

class SocBasedPowerDistributor

Bases: *simses.logic.power_distribution.power_distributor.PowerDistributor*

The SocBasedPowerDistributor calculates the power share to the systems via an inverse distance weighting algorithm according to the system SOC. In charge case, the system with lower SOC receives a higher load - in discharge case vice versa.

get_power_for (*power*, *state*)

set (*time*, *states*, *power*)

class PowerDistributorState(*sys_id*, *soh=0.0*, *soc=0.0*, *storage_technology=None*, *max_charge_power=0.0*, *max_discharge_power=0.0*)

Bases: `object`

PowerDistributorStates contains information for power distribution algorithms.

property capacity

property max_charge_power

property max_discharge_power

property rebalance_charge_power

property rebalance_discharge_power

property soc

property soh

static sort_by_soc(*data*, *descending=False*)

In-place sorting of list with PowerDistributorState objects by soc (ascending by default)

Parameters

- **descending** – reverse sorting of data list, default: False
- **data** – list of TimeValue objects

static sort_by_soh(*data*, *descending=True*)

In-place sorting of list with PowerDistributorState objects by soh (descending by default)

Parameters

- **descending** – reverse sorting of data list, default: True
- **data** – list of TimeValue objects

property storage_technology

property sys_id

property time_delta

class TechnologyBasedPowerDistributor(*priorities*, *storage_systems*)

Bases: *simses.logic.power_distribution.power_distributor.PowerDistributor*

TechnologyBasedPowerDistributor distributes the power depend on the current state and storage technology used. A priority list for the technology is used for the order of execution. In addition, TechnologyBasedPowerDistributor tries to rebalance the SOC of the various storage devices if possible.

close()

get_power_for(*power_target*, *state*)

set (*time*, *states*, *power*)

Module contents

simses.logic.test package

Submodules

create_state (*soc*)

test_power (*soc_high*, *soc_low*, *uut*, *power*)

uut ()

create_ems_config ()

create_file ()

create_general_config ()

test_next (*time*, *soc*, *power_offset*, *result*, *uut*)

uut ()

create_general_config ()

create_load_file ()

create_pv_file ()

test_next (*time*, *soc*, *result*, *uut*)

uut ()

create_state (*soc*)

test_power (*soc_high*, *soc_low*, *uut*, *power*)

uut ()

Module contents

Module contents

2.5 Technology

2.5.1 simses.technology package

Subpackages

simses.technology.hydrogen package

Subpackages

simses.technology.hydrogen.control package

Subpackages

simses.technology.hydrogen.control.pressure package

Submodules

class IdealVarCathodePressureController(*config*)

Bases: *simses.technology.hydrogen.control.pressure.pressure_controller.PressureController*

This pressure controller controls the cathode pressure at a disired level and keeps the anode pressure at ambient level

calculate_n_h2_in(*pressure_anode*, *n_h2_used*)

calculate_n_h2_out(*pressure_cathode*, *n_h2_prod*, *timestep*, *pressure_factor*)

calculates the necessary hydrogen outflow to reach the desire pressure relative to the actual hydrogen production and cathode pressure

input: pressure cathode in barg, pressure cathode desire in barg, h2 production in mol/s, timestep in s, pressure factor in bar/mol

output: h2 outflow in mol/s

calculate_n_o2_in(*pressure_cathode*, *n_o2_used*)

calculate_n_o2_out(*pressure_anode*, *n_o2_prod*, *timestep*)

class NoPressureController

Bases: *simses.technology.hydrogen.control.pressure.pressure_controller.PressureController*

calculate_n_h2_in(*pressure_anode*, *n_h2_used*)

calculate_n_h2_out(*pressure_cathode*, *n_h2_prod*, *timestep*, *pressure_factor*)

calculate_n_o2_in(*pressure_cathode*, *n_o2_used*)

calculate_n_o2_out(*pressure_anode*, *n_o2_prod*, *timestep*)

class PressureController

Bases: `abc.ABC`

This controller controls the pressure of anode and cathode side of the electrolyzer like a control valve

abstract calculate_n_h2_in(*pressure_anode*, *n_h2_used*)

abstract calculate_n_h2_out(*pressure_cathode*, *n_h2_prod*, *timestep*, *pressure_factor*)

abstract calculate_n_o2_in(*pressure_cathode*, *n_o2_used*)

abstract calculate_n_o2_out(*pressure_anode*, *n_o2_prod*, *timestep*)

Module contents

simses.technology.hydrogen.control.thermal package

Submodules

class IdealVarFlowThermalController(*config*, *heat_capacity_stack*)

Bases: `simses.technology.hydrogen.control.thermal.thermal_controller.ThermalController`

This controller controls the temperature of the EL-stack by varing the input temperaure of the feeding water and in the second step adapt the mass flow of the water in order to reach the desired temperature

calculate(*stack_temperature*, *heat_stack*, *timestep*, *current_dens*)

check_control_status(*current_dens*, *timestep*)

get_delta_water_temp_in()

get_h2o_flow()

get_heat_control_on()

class NoThermalController

Bases: `simses.technology.hydrogen.control.thermal.thermal_controller.ThermalController`

calculate(*stack_temperature*, *heat_stack*, *timestep*, *current_dens*)

get_delta_water_temp_in()

get_h2o_flow()

get_heat_control_on()

class ThermalController

Bases: `abc.ABC`

This controller controls the temperature of the EL-stack by setting new values for the mass flow and the temperature of the water running through the stack. It is asumed that the water temperature coming out of the stack equals the stack temperature

abstract calculate(*stack_temperature*, *heat_stack*, *timestep*, *current_dens*)

abstract get_delta_water_temp_in()

abstract get_h2o_flow()

abstract get_heat_control_on()

class VarFlowThermalController(*config*, *min_water_flow_stack*, *heat_capacity*)

Bases: simses.technology.hydrogen.control.thermal.thermal_controller.ThermalController

This controller controls the temperature of the EL-stack by varing the input temperaure of the feeding water and in the second step adapt the mass flow of the water in order to reach the desired temperature

calculate(*stack_temperature*, *heat_stack*, *timestep*, *current_dens*)

get_delta_water_temp_in()

get_h2o_flow()

Module contents

Submodules

class HydrogenManagementSystem(*min_power_electroyzer*, *max_power_electrolyzer*, *max_power_fuel_cell*, *config*)

Bases: object

close()

Closing all resources

update(*time*, *state*)

Updating power and fulfillment factor in order to check restrictions

Module contents

simses.technology.hydrogen.electrolyzer package

Subpackages

simses.technology.hydrogen.electrolyzer.degradation package

Subpackages

simses.technology.hydrogen.electrolyzer.degradation.calendar package

Submodules

class CalendarDegradationModel

Bases: `abc.ABC`

Degradation Model for the calendaric aging of the Elektrolyzer.

calculate_degradation(*state*)

abstract calculate_exchange_current_dens_decrease(*state*)

update the calendric decrease of the exchange current denisty of the electrolyzer :param state: :return:

abstract calculate_resistance_increase(*state*)

update the calendary internal resistance increase of a electrolyzer :param state: :type state: HydrogenState

abstract close()

Closing all resources in calendar degradation model

abstract get_exchange_current_dens_decrease()

get the updated caledric exchange current density decrease :return:

abstract get_resistance_increase()

get the updated calendric resistance increase :returns: resistance increase in [p.u.] :rtype: float

abstract reset(*state*)

resets all values within a calendar degradation model, if battery is replaced :param battery_state: :type battery_state: LithiumIonState; Current BatteryState of the lithium_ion.

class NoCalendarDegradationModel

Bases: *simses.technology.hydrogen.electrolyzer.degradation.calendar.calendar_degradation.CalendarDegradationModel*

calculate_exchange_current_dens_decrease(*state*)

calculate_resistance_increase(*state*)

close()

get_exchange_current_dens_decrease()

get_resistance_increase()

reset(*state*)

class CalendarDegradationPemElMultiDimAnalyitic(*general_config*)

Bases: *simses.technology.hydrogen.electrolyzer.degradation.calendar.calendar_degradation.CalendarDegradationModel*

Calendaric degradation model for PemElectrolyzerMultiDimAnalyitc decreases the exchange current density in dependency of the operation time

calculate_exchange_current_dens_decrease(*state*)

Calculation of exchange current density decrease dependent on time the electrolyzer cell is in operation. based on paper: "Polymer electrolyte membrane water electrolysis: Restraining degradation in the presence of fluctuating power" by Rakousky, Christoph year: 2017 :param state: :return: none

calculate_resistance_increase (*state*)

close ()

get_exchange_current_dens_decrease ()

get_resistance_increase ()

reset (*state*)

Module contents

simses.technology.hydrogen.electrolyzer.degradation.cyclic package

Submodules

class CyclicDegradationModel

Bases: `abc.ABC`

Degradation Model for the cyclic aging of the Electrolyzer.

calculate_degradation (*time*, *state*)

abstract calculate_exchange_current_dens_decrerase (*state*)

update the cyclic decrease of the exchange current denisty of the electrolyzer :param state: :return:

abstract calculate_resistance_increase (*time*, *state*)

update the cyclic internal resistance increase of a electrolyzer :param time: :type time: Float :param state: :type state: HydrogenState

abstract close ()

Closing all resources in calendar degradation model

abstract get_exchange_current_dens_decrease ()

get the updated caledric exchange current density decrease :return:

abstract get_resistance_increase ()

get the updated calendary resistance increase :returns: resistance increase in [p.u.] :rtype: float

abstract reset (*hydrogen_state*)

resets all values within a calendar degradation model, if electrolyzer is replaced :param hydrogen_state: :type hydrogen_state: HydrogenState

class NoCyclicDegradationModel

Bases: *simses.technology.hydrogen.electrolyzer.degradation.cyclic.cyclic_degradation.CyclicDegradationModel*

calculate_exchange_current_dens_decrerase (*state*)

calculate_resistance_increase (*time*, *state*)

close ()

get_exchange_current_dens_decrease ()

get_resistance_increase ()

reset (*hydrogen_state*)

class CyclicDegradationPemElMultiDimAnalytic

Bases: *simses.technology.hydrogen.electrolyzer.degradation.cyclic.cyclic_degradation.CyclicDegradationModel*

calculate_exchange_current_dens_decrerase (*state*)

No cyclic exchange_current_density_decrease

calculate_resistance_increase (*time*, *state*)

Calculation of resistance increase dependent on currentdensity provided to the electrolyzer cell recovery effect: all time exponential based on paper: "Polymer electrolyte membrane water electrolysis: Restraining degradation in the presence of fluctuating power" by Rakousky, Christoph year: 2017 :param time: :param state: :return: none

close ()

get_exchange_current_dens_decrease ()

get_resistance_increase ()

reset (*hydrogen_state*)

class CyclicDegradationPemElMultiDimAnalyticPtlCoating

Bases: *simses.technology.hydrogen.electrolyzer.degradation.cyclic.cyclic_degradation.CyclicDegradationModel*

Anode Ti-PTL of Electrolyzer cell is coated with thin Pt. This avoids the cyclic degradation of the resistance. Therefore this degradation model does not have cyclic degradation.

calculate_exchange_current_dens_decrerase (*state*)

No cyclic exchange_current_density_decrease

calculate_resistance_increase (*time*, *state*)

" No cyclic resistance increase

close ()

get_exchange_current_dens_decrease ()

get_resistance_increase ()

reset (*state*)

Module contents

Submodules

class DegradationModel(*cyclic_degradation_model*, *calendar_degradation_model*)

Bases: `abc.ABC`

Model for the degradation_model_el behavior of the electrolyzer by analysing the resistance increase and exchange current density.

calculate_degradation(*time*, *state*)

Calculates degradation parameters of the specific electrolyzer

Parameters

- **time** (*float*) – Current timestamp.
- **state** (HydrogenState) – Current state of the Electrolyzer.

abstract calculate_soh(*state*)

Calculates the SOH of the electrolyzer

Parameters **state** (HydrogenState) – Current state of health of the electrolyzer.

close()

Closing all resources in degradation_model_el model

abstract get_soh_el()

update(*time*, *state*)

Updating the resistance and exchange current density of the electrolyzer through the degradation_model_el model.

Parameters

- **time** (*float*) – Current timestamp.
- **state** (HydrogenState) – Current state of the hydrogen storage system.

class NoDegradationModel

Bases: *simses.technology.hydrogen.electrolyzer.degradation.degradation_model.DegradationModel*

calculate_soh(*state*)

get_soh_el()

class PemElMultiDimAnalyticDegradationModel(*electrolyzer*, *config*, *general_config*)

Bases: *simses.technology.hydrogen.electrolyzer.degradation.degradation_model.DegradationModel*

calculate_soh(*state*)

get_soh_el()

class PemElMultiDimAnalyticDegradationModelPtlCoating(*electrolyzer*, *config*, *general_config*)

Bases: *simses.technology.hydrogen.electrolyzer.degradation.degradation_model.DegradationModel*

calculate_soh(*state*)

get_soh_el()

Module contents

simses.technology.hydrogen.electrolyzer.pressure package

Submodules

class PressureModel

Bases: abc.ABC

PressureModelEl calculates the pressure development at the cathode of the electrolyzer in dependency of the hydrogenproductionrate, the hydrogenoutflow and the temperature

abstract calculate(*time*, *state*)

close()

Closing all resources in pressure model

abstract get_h2_outflow()

abstract get_h2o_outflow_anode()

abstract get_h2o_outflow_cathode()

abstract get_o2_outflow()

abstract get_pressure_anode()

abstract get_pressure_cathode()

update(*time*, *state*)

class AlkalinePressureModel(*config*)

Bases: *simses.technology.hydrogen.electrolyzer.pressure.abstract_model.PressureModel*

calculate(*time*, *state*)

get_h2_outflow()

get_h2o_outflow_anode()

get_h2o_outflow_cathode()

get_o2_outflow()

get_pressure_anode()

get_pressure_cathode()

class NoPressureModel

Bases: *simses.technology.hydrogen.electrolyzer.pressure.abstract_model.PressureModel*

calculate(*time*, *state*)

get_h2_outflow()

get_h2o_outflow_anode()

get_h2o_outflow_cathode()

get_o2_outflow()

get_pressure_anode()

get_pressure_cathode()

class VarCathodePressureModel(*electrolyzer*, *config*)

Bases: *simses.technology.hydrogen.electrolyzer.pressure.abstract_model.PressureModel*

calculate(*time*, *state*)

get_h2_outflow()

get_h2o_outflow_anode()

get_h2o_outflow_cathode()

get_o2_outflow()

get_pressure_anode()

get_pressure_cathode()

Module contents

simses.technology.hydrogen.electrolyzer.stack package

Subpackages

simses.technology.hydrogen.electrolyzer.stack.alkaline package

Submodules

class AlkalineElectricalModel(*pressure_model*, *fluid_model*, *electrolyzer_data_config*, *parameters*)

Bases: object

calculate_bubble_rate_coverage(*current*, *stack_temperature*)

get_cell_voltage(*current*, *state*)

get_current(*power_cell*, *state*)

get_geometric_area_cell()

get_rev_voltage(*stack_temperature*)
Returns reversible Cell Voltage at standard pressure depending on stack temperature in K

class AlkalineElectrolyzer(*electrolyzer_maximal_power*, *electrolyzer_data_config*)
Bases: `simses.technology.hydrogen.electrolyzer.stack.stack_model.ElectrolyzerStackModel`

An Alkaline Electrolyzer is a special type of Electrolyzer

calculate(*power*, *state*)

close()

get_current()

get_current_density()

get_efficiency_curve(*hydrogen_state*)

get_geom_area_stack()

get_heat_capacity_stack()

get_hydrogen_production()

get_nominal_current_density()

get_nominal_stack_power()

get_number_cells()

get_oxygen_production()

get_partial_pressure_h2()

get_partial_pressure_o2()

get_reference_voltage_eol(*resistance_increase*, *exchange_current_decrease*)

get_sat_pressure_h2o()

get_thermal_resistance_stack()

get_voltage()

get_water_in_stack()

get_water_use()

class AlkalineFluidModel
Bases: `object`

get_h2_generation_cell (*current_cell*)

get_h2o_use_cell (*current_cell*)

get_molarity (*stack_temperature*)

get_o2_generation_cell (*current_cell*)

class ParametersAlkalineElectrolyzerFit (*electrolyzer_data_config*)
Bases: `object`

property a1

property a2

property aa

property ac

property ba

property bc

property ca

property cc

property r1

property r2

property r3

property zirfon1

property zirfon2

property zirfon3

class AlkalinePressureModel
Bases: `object`

get_aqueous_vapour_pressure_koh (*molarity*, *stack_temperature*)

get_partial_pressure_h2 (*molarity*, *stack_temperature*)

get_partial_pressure_o2 (*molarity*, *stack_temperature*)

get_sat_pressure_h2o (*molarity*, *stack_temperature*)

get_vapour_pressure_pure_water (*stack_temperature*)

Module contents

simses.technology.hydrogen.electrolyzer.stack.pem_analytic package

Submodules

class PemElectrolyzerEfficiencyCurves(*electrical_model*, *membrane_model*, *pressure_model*, *fluid_modell*)

Bases: `object`

calculate_efficiency_curves(*hydrogen_state*, *start=0.01*, *stop=3.01*, *step=0.01*)

class PemElectricalModel(*membrane_model*, *pressure_model*, *parameters*)

Bases: `object`

P_H2_REF = 1

P_O2_REF = 1

R_ELE = 0.096

get_cell_voltage(*current_dens*, *state*)

get_current_density(*power_dens_cell*, *state*)

get_ohm_resistance(*temperature*)

Returns ohmic resistance of Cell in Ohm cm^2

get_rev_voltage(*temperature*)

Returns reversible Cell Voltage dependent on stack temperature in °C

get_rev_voltage_for_current_calc(*temperature*)

Returns reversible Cell Voltage dependent on stack temperature in °C, but only for calculation of current density, use of correction parameters!!!!

class PemElectrolyzerMultiDimAnalytic(*electrolyzer_maximal_power*, *electrolyzer_data_config*)

Bases: *simses.technology.hydrogen.electrolyzer.stack.stack_model.ElectrolyzerStackModel*

An PEM-Electrolyzer is a special typ of an Electrolyzer

calculate(*power*, *state*)

close()

get_current()

get_current_density()

get_efficiency_curve(*hydrogen_state*)

get_geom_area_stack()

get_heat_capacity_stack()

get_hydrogen_production()

get_nominal_current_density()

get_nominal_stack_power()

get_number_cells()

get_oxygen_production()

get_partial_pressure_h2()

get_partial_pressure_o2()

get_reference_voltage_eol(*resistance_increase*, *exchange_current_decrease*)

return cell voltage of electrolyzer at 2 A/cm^2, p_anode = 0 barg, p_cathode = 0 barg and temperature = 80°C this cell voltage is needed for the calculation of the SOH of the electrolyzer :return: cell voltage in V

get_sat_pressure_h2o()

get_thermal_resistance_stack()

get_voltage()

get_water_in_stack()

get_water_use()

class PemElectrolyzerMultiDimAnalyticPtlCoating(*electrolyzer_maximal_power*, *electrolyzer_data_config*)

Bases: `simses.technology.hydrogen.electrolyzer.stack.pem_analytic.electrolyzer_model.PemElectrolyzerMultiDimAnalytic`

PEM Electrolyzer with Pt-coated Ti-PTL at anode site, which prevents the cyclic increase of its resistance

class PemFluidModel(*membrane_model*)

Bases: `object`

get_h2_generation_cell(*current_cell*)

get_h2_net_cathode(*state*, *current_cell*)

get_h2o_net_use_cell(*state*, *current_cell*)

get_h2o_use_cell(*current_cell*)

get_o2_generation_cell(*current_cell*)

get_o2_net_anode(*state*, *current_cell*)

class PemMembraneModel(*pressure_model*, *geom_area_cell=1225*, *thickness=0.02*, *humidification=25.0*)

Bases: `object`

DIFF_COEF_H2 = 4.65e-11

DIFF_COEF_O2 = 2e-11

DP_COEF_H2 = 2e-11

EPS = 2.220446049250313e-16

GEOM_AREA_CELL = 1225

WATER_DRAG_COEFF = 0.27

get_geometric_area_cell()

get_h2_permeation_for_cell(*state*, *current_cell*)
permeation of hydrogen through membrane due to diffusion because of differential partial pressures :param part_pressure_h2: :param part_pressure_o2:

Returns mol/s

Return type float

get_h2o_permeation_for_cell(*current_cell*)

get_o2_permeation_for_cell(*state*, *current_cell*)
permeation of oxygen through membrane due to diffusion because of differential partial pressures

Parameters **part_pressure_o2** –

resistance(*stack_temperature*)

class ParametersPemElectrolyzerMultiDimAnalytic(*electrolyzer_data_config*)
Bases: `object`

property p10

property p11

property p20

property q1

property q10

property q11

property q12

property q13

property q14

property q15

property q17

property q3

property q4

property q5

property q6

property q7

property q9

class PemPressureModel(*parameters*)

Bases: `object`

A_ANODE = 2.8

A_CATHODE = 2.4

get_partial_pressure_h2(*state*, *current_dens_cell*)

get_partial_pressure_h2_for_current_calc(*state*, *current_dens_cell*)

get_partial_pressure_o2(*state*, *current_dens_cell*)

get_partial_pressure_o2_for_current_calc(*state*, *current_dens_cell*)

get_sat_pressure_h2o(*stack_temperature*)

get_sat_pressure_h2o_for_current_calc(*stack_temperature*)

Module contents

Submodules

class NoElectrolyzer

Bases: *simses.technology.hydrogen.electrolyzer.stack.stack_model.ElectrolyzerStackModel*

calculate(*power*, *state*)

close()

get_current()

get_current_density()

get_geom_area_stack()

get_heat_capacity_stack()

get_hydrogen_production()

get_nominal_current_density()

get_nominal_stack_power()

get_number_cells()

get_oxygen_production()

get_partial_pressure_h2()

get_partial_pressure_o2()

get_reference_voltage_eol(*resistance_increase, exchange_currentdensity_decrease*)

get_sat_pressure_h2o()

get_thermal_resistance_stack()

get_voltage()

get_water_in_stack()

get_water_use()

class PemElectrolyzer(*electrolyzer_maximal_power, electrolyzer_data_config*)

Bases: `simses.technology.hydrogen.electrolyzer.stack.stack_model.ElectrolyzerStackModel`

An PEM-Electrolyzer is a special typ of an Electrolyzer

calculate(*power, state*)

close()

get_current()

get_current_density()

get_geom_area_stack()

get_heat_capacity_stack()

get_hydrogen_production()

get_nominal_current_density()

get_nominal_stack_power()

get_number_cells()

get_oxygen_production()

get_partial_pressure_h2()

get_partial_pressure_o2()

get_reference_voltage_eol(*resistance_increase, exchange_currentdensity_decrease*)

get_sat_pressure_h2o()

get_thermal_resistance_stack()

get_voltage()

get_water_in_stack()

get_water_use()

class ElectrolyzerStackModel

Bases: `abc.ABC`

abstract calculate(*power*, *state*)
Calculates current, voltage and hydrogen generation based on input power

Parameters

- **power** (*input power in W*) –
- **temperature** (*temperature of electrolyzer in K*) –
- **pressure_anode** (*relative pressure on anode side of electrolyzer in barg (relative to 1 bar)*) –
- **pressure_cathode** (*relative pressure on cathode side of electrolyzer in barg (relative to 1 bar)*) –

abstract close()

abstract get_current()
return electrical current of the electrolyzer stack in A

abstract get_current_density()
return electrical current of the electrolyzer stack in A cm-2

abstract get_geom_area_stack()

abstract get_heat_capacity_stack()

abstract get_hydrogen_production()
Return of total hydrogen generation of the stack in mol/s

abstract get_nominal_current_density()

abstract get_nominal_stack_power()

abstract get_number_cells()

abstract get_oxygen_production()
Return of total oxygen generation of the stack in mol/s

abstract get_partial_pressure_h2()

abstract get_partial_pressure_o2()

abstract get_reference_voltage_eol(*resistance_increase*, *exchange_currentdensity_decrease*)
return voltage at defined operation point for state of degradation

Returns

abstract get_sat_pressure_h2o()

abstract get_thermal_resistance_stack()

abstract get_voltage()
Return of electrical voltage of electrolyzer stack in V

abstract get_water_in_stack()

abstract get_water_use()
Return of water use of electrolyzer stack at anode side

update(*power, state*)
Updates hydrogen states that are corrolated with the electrolyzer such as current, voltage, hydrogen production, water use and temperature

Parameters state –

Module contents

simses.technology.hydrogen.electrolyzer.thermal package

Submodules

class ThermalModel
Bases: abc.ABC

abstract calculate(*time, state, pressure_cathode_0, pressure_anode_0*)

abstract calculate_pump_power(*water_flow_stack*)

close()
Closing all resources in thermal model

abstract get_convection_heat()

abstract get_power_water_heating()

abstract get_pump_power()

abstract get_temperature()

abstract get_water_flow_stack()

update(*time, state, pressure_cathode_0, pressure_anode_0*)
Updating temperature of electrolyzer (°C) stack in hydrogen state

class SimpleThermalModelAlkaline(*electrolyzer, water_heating, pump, temperature, config*)
Bases: *simses.technology.hydrogen.electrolyzer.thermal.abstract_model.ThermalModel*

This model functions at Electrolyzer Level. This model calculates the temperaturechange in the electrlyzer stack the elelctrolyzer is represented by a area element

calculate(*time, state, pressure_cathode_0, pressure_anode_0*)

calculate_pump_power(*water_flow_stack*)

get_convection_heat()

get_power_water_heating()

get_pump_power()

get_temperature()

get_water_flow_stack()

class NoThermalModel

Bases: *simses.technology.hydrogen.electrolyzer.thermal.abstract_model.ThermalModel*

This model functions at the Storage Technology Level. This model treats the entire Storage Technology as 1 lump. Current version sets temperature of Storage Technology to 298.15 K and treats it as constant

calculate(*time*, *state*, *pressure_cathode_0*, *pressure_anode_0*)

calculate_pump_power(*water_flow_stack*)

calculate_tube_pressure_loss(*water_flow_stack*)

get_convection_heat()

get_power_water_heating()

get_pump_power()

get_temperature()

get_tube_pressure_loss()

get_water_flow_stack()

set_temperature(*new_temperature*)

class SimpleThermalModel(*electrolyzer*, *water_heating*, *pump*, *temperature*, *config*)

Bases: *simses.technology.hydrogen.electrolyzer.thermal.abstract_model.ThermalModel*

This model functions at Electrolyzer Level. This model calculates the temperaturechange in the electrlyzer stack the elelctrolyzer is represented by a area element

calculate(*time*, *state*, *pressure_cathode_0*, *pressure_anode_0*)

calculate_pump_power(*water_flow_stack*)

get_convection_heat()

get_power_water_heating()

get_pump_power()

get_temperature()

get_water_flow_stack()

Module contents

Submodules

class ElectrolyzerFactory(*config*)
Bases: object

close()

create_degradation_model(*electrolyzer*)

create_electrolyzer_stack(*electrolyzer, electrolyzer_maximal_power*)

create_pressure_model(*electrolyzer*)

create_state(*system_id, storage_id, temperature, electrolyzer, pressure_model*)

create_thermal_model(*electrolyzer, water_heating, pump, temperature*)

class Electrolyzer(*system_id, storage_id, electrolyzer, max_power, temperature, config, data_handler*)
Bases: object

Electrolyzer is the top level class incorporating a ElectrolyzerStackModel, a ThermalModel and a PressureModel. The specific classes are instantiated within the ElectrolyzerFactory.

close()

get_auxiliaries()

property state

update(*time, power*)
Updates hydrogen states that are corrolated with the electrolyzer such as current, voltage, hydrogen production, water use and temperature

Parameters

- **time** –
- **hydrogen_state** –
- **power** –

Module contents

simses.technology.hydrogen.fuel_cell package

Subpackages

simses.technology.hydrogen.fuel_cell.pressure package

Submodules

class NoPressureModel

Bases: *simses.technology.hydrogen.fuel_cell.pressure.pressure_model.PressureModel*

calculate(*time, state*)

get_h2_inflow()

get_o2_inflow()

get_pressure_anode_fc()

get_pressure_cathode_fc()

class PressureModel

Bases: abc.ABC

PressureModelEl calculates the pressure development at the cathode of the electrolyzer in dependency of the hydrogenproductionrate, the hydrogenoutflow and the temperature

abstract calculate(*time, state*)

close()

Closing all resources in pressure model

abstract get_h2_inflow()

abstract get_o2_inflow()

abstract get_pressure_anode_fc()

abstract get_pressure_cathode_fc()

update(*time, state*)

Module contents

simses.technology.hydrogen.fuel_cell.stack package

Submodules

class JupiterFuelCell(*fuel_cell_maximal_power, fuel_cell_data_config*)

Bases: *simses.technology.hydrogen.fuel_cell.stack.stack_model.FuelCellStackModel*

A Jupiter-Fuelcell is a special typ of a PEM-Fuelcell with an open cathode from SFC Energy: https://www.efoy-pro.com/efoy-pro/efoy-jupiter-2-5/

calculate(*power*)

close()

get_current()

get_current_density()

get_geom_area_stack()

get_heat_capactiy_stack()

get_hydrogen_consumption()

get_nominal_stack_power()

get_number_cells()

get_voltage()

class NoFuelCell

Bases: *simses.technology.hydrogen.fuel_cell.stack.stack_model.FuelCellStackModel*

An No-Fuelcell is a special typ of a Fuelcell

calculate(*power*)

close()

get_current()

get_geom_area_stack()

get_heat_capactiy_stack()

get_hydrogen_consumption()

get_nominal_stack_power()

get_number_cells()

get_voltage()

class PemFuelCell(*fuel_cell_maximal_power*, *fuel_cell_data_config*)

Bases: *simses.technology.hydrogen.fuel_cell.stack.stack_model.FuelCellStackModel*

An PEM-Fuelcell is a special typ of a Fuelcell

calculate(*power*)

close()

get_current()

get_geom_area_stack()

get_heat_capactiy_stack()

get_hydrogen_consumption()

get_nominal_stack_power()

get_number_cells()

get_voltage()

class FuelCellStackModel

Bases: `abc.ABC`

abstract calculate(*power*)

Calculates current, voltage and hydrogen consumption based on input power :param power: :type power: input power in W

abstract close()

abstract get_current()

return electrical current in A

get_current_density()

return electrical current of the electrolyzer stack in A cm-2

abstract get_geom_area_stack()

abstract get_heat_capactiy_stack()

abstract get_hydrogen_consumption()

Return of hydrogen consumption in mol/s

abstract get_nominal_stack_power()

abstract get_number_cells()

abstract get_voltage()

Return of electrical voltage in V

update(*power, state*)

Updates current, voltage and hydrogen flow of hydrogen state

Parameters

- **power** –
- **state** –

Module contents

simses.technology.hydrogen.fuel_cell.thermal package

Submodules

class NoThermalModel

Bases: *simses.technology.hydrogen.fuel_cell.thermal.thermal_model.ThermalModel*

This model functions at the Storage Technology Level. This model treats the entire Storage Technology as 1 lump. Current version sets temperature of Storage Technology to 298.15 K and treats it as constant

calculate(*time, state*)

get_power_water_heating()

get_pump_power ()

get_temperature ()

get_water_flow_stack ()

class SimpleThermalModel (*temperature*)

Bases: *simses.technology.hydrogen.fuel_cell.thermal.thermal_model.ThermalModel*

Simple Thermal Model with assuming a direct dependency to the fuel cell current based on information of Ballard's "Product Manual and Integration Guide Integrating the 1020ACS into a System", 2016

calculate (*time*, *state*)

get_power_water_heating ()

get_pump_power ()

get_temperature ()

get_water_flow_stack ()

class ThermalModel

Bases: abc.ABC

abstract calculate (*time*, *state*)

close ()

Closing all resources in thermal model

abstract get_power_water_heating ()

abstract get_pump_power ()

abstract get_temperature ()

abstract get_water_flow_stack ()

update (*time*, *state*)

Updating temperature of electrolyzer stack in hydrogen state

Module contents

Submodules

class FuelCellFactory (*config*)

Bases: object

close ()

create_fuel_cell_stack (*fuel_cell*, *fuel_cell_maximal_power*)

create_pressure_model (*fuel_cell*)

create_state (*system_id*, *storage_id*, *temperature*, *fuel_cell*)

create_thermal_model (*fuel_cell*, *temperature*)

class FuelCell (*system_id*, *storage_id*, *fuel_cell*, *max_power*, *temperature*, *config*, *data_handler*)

Bases: `object`

FuelCell is the top level class incorporating a FuelCellStackModel, a ThermalModel and a PressureModel. The specific classes are instantiated within the FuelCellFactory.

close ()

get_auxiliaries ()

property state

update (*time*, *power*)

Updates current, voltage and hydrogen flow of hydrogen state

Parameters

- **time** –
- **state** –
- **power** –

Module contents

simses.technology.hydrogen.hydrogen_storage package

Subpackages

simses.technology.hydrogen.hydrogen_storage.pipeline package

Submodules

class Pipeline

Bases: *simses.technology.hydrogen.hydrogen_storage.hydrogen_storage.HydrogenStorage*

abstract calculate_injected_hydrogen (*time_diff*, *hydrogen_outflow*)

abstract get_injected_hydrogen ()

abstract get_tank_pressure ()

class SimplePipeline (*storage_pressure*)

Bases: *simses.technology.hydrogen.hydrogen_storage.pipeline.pipeline.Pipeline*

calculates total mass of produced and injected hydrogen in kg

calculate_injected_hydrogen (*time_diff*, *hydrogen_outflow*)

calculate_soc(*time_diff, hydrogen_net_flow*)

close()

get_capacity()

get_injected_hydrogen()

get_soc()

get_tank_pressure()

Module contents

simses.technology.hydrogen.hydrogen_storage.pressuretank package

Submodules

class PressureTank(*capacity, max_pressure, soc*)
Bases: *simses.technology.hydrogen.hydrogen_storage.hydrogen_storage.HydrogenStorage*

calculate_soc(*time_difference, hydrogen_net_flow*)

close()

get_capacity()

get_soc()

get_tank_pressure()

Module contents

Submodules

class HydrogenStorage
Bases: abc.ABC

abstract calculate_soc(*time_diff, hydrogen_net_flow*)

abstract close()

get_auxiliaries()

abstract get_capacity()

abstract get_soc()

abstract get_tank_pressure()

update_from(*time_difference, electrolyzer_state, fuel_cell_state*)

Parameters

- **time** –
- **electrolyzer_state** –
- **fuel_cell_state** –

Module contents

simses.technology.hydrogen.test package

Submodules

create_electrolyzer_state_from (*electrolyzer*)

test_calculate_current (*uut, power, temperature, pressure_anode, pressure_cathode, current_result*)

test_calculate_geq_current (*uut, power, temperature, pressure_anode, pressure_cathode, current_result*)

test_calculate_hydrogen_flow (*uut, power, temperature, pressure_anode, pressure_cathode, h2_flow_result*)

test_calculate_hydrogen_flow_geq (*uut, power, temperature, pressure_anode, pressure_cathode, h2_flow_result*)

test_calculate_voltage (*uut, power, temperature, pressure_anode, pressure_cathode, voltage_result*)

test_calculate_water_use (*uut, power, temperature, pressure_anode, pressure_cathode, get_water_use_result*)

uut (*nominal_power*)

Module contents

Submodules

class HydrogenFactory (*config*)

Bases: `object`

close ()

create_hydrogen_management_system (*electrolyzer_minimal_power, electrolyzer_maximal_power, fuel_cell_maximal_power*)

create_hydrogen_state_from (*system_id, storage_id*)

create_hydrogen_storage (*storage, capacity, max_pressure*)

class Hydrogen(*data_export*, *fuel_cell*, *fuel_cell_maximal_power*, *electrolyzer*, *electrolyzer_maximal_power*, *storage*, *capacity*, *max_pressure*, *temperature*, *system_id*, *storage_id*, *config*)

Bases: `simses.technology.storage.StorageTechnology`

Hydrogen is the top level class for coordinating all technologies related to hydrogen, i.e. electrolyzer, fuel cells and storage tank. It provides a management system for taking limitations into consideration. The hydrogen class requires always an electrolyzer, a fuel cell and a storage. In cases this circle is unwanted, dummy classes exists doing nothing, e.g. NoElectrolyzer or NoFuelCell.

close()

property convection_coefficient

distribute_and_run(*time*, *current*, *voltage*)

get_auxiliaries()

get_system_parameters()

property mass

property specific_heat

property state

property surface_area

property volume

wait()

Module contents

simses.technology.lithium_ion package

Subpackages

simses.technology.lithium_ion.battery_management_system package

Submodules

class BatteryManagementSystem(*cell_type*, *battery_config*)

Bases: `object`

BatteryManagementSystem class

close()

Closing all resources in lithium_ion management system

update(*time*, *battery_state*, *power_target*)

Updating current of lithium_ion state in order to comply with cell type restrictions

Module contents

simses.technology.lithium_ion.cell package

Submodules

class NRELDummyCell(*voltage, capacity, soh, battery_config*)
Bases: *simses.technology.lithium_ion.cell.type.CellType*

An GenericCell is a special cell type and inherited by CellType

a_0 = 0.8525724500396755

close()

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)

class GenericCell(*voltage, capacity, soh, battery_config*)
Bases: *simses.technology.lithium_ion.cell.type.CellType*

An GenericCell is a special cell type and inherited by CellType

close()

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)

class SonyLFP(*voltage, capacity, soh, battery_config, battery_data_config*)
Bases: *simses.technology.lithium_ion.cell.type.CellType*

Source SONY_US26650FTC1_Product Specification and Naumann, Maik. Techno-economic evaluation of stationary lithium_ion energy storage systems with special consideration of aging. PhD Thesis. Technical University Munich, 2018.

close()

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)
Parameters build with ocv fitting

class DaimlerLMO(*voltage, capacity, soh, battery_config, battery_data_config*)
Bases: *simses.technology.lithium_ion.cell.type.CellType*

A LMO (Daimler LMO) is a special cell type and inherited by CellType

close()

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)

class PanasonicNCA(*voltage, capacity, soh, battery_config, battery_data_config*)
Bases: *simses.technology.lithium_ion.cell.type.CellType*

An NCA (PanasonicNCR) is a special cell type and inherited by CellType

close()

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)
Parameters build with ocv fitting

class MolicelNMC(*voltage*, *capacity*, *soh*, *battery_config*, *battery_data_config*)
Bases: `simses.technology.lithium_ion.cell.type.CellType`

An NMC (NMC_Molicel) is a special cell type and inherited by CellType

close()

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)
Parameters build with ocv fitting

class Samsung78AhNMC(*voltage*, *capacity*, *soh*, *battery_config*)
Bases: `simses.technology.lithium_ion.cell.type.CellType`

Characterisation using field data: Master Thesis by Felix Müller

close()

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)

class Samsung94AhNMC(*voltage*, *capacity*, *soh*, *battery_config*, *battery_data_config*)
Bases: `simses.technology.lithium_ion.cell.type.CellType`

A NMC (Samsung NMC 94Ah) is a special cell type and inherited by CellType

close()

get_capacity(*battery_state*)

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)

class SanyoNMC(*voltage*, *capacity*, *soh*, *battery_config*, *battery_data_config*)
Bases: `simses.technology.lithium_ion.cell.type.CellType`

An NMC (Sanyo UR18650E) is a special cell type and inherited by CellType

close()

get_internal_resistance(*battery_state*)

get_open_circuit_voltage(*battery_state*)
Parameters build with ocv fitting

class CellType(*voltage*, *capacity*, *soh*, *electrical_props*, *thermal_props*, *cell_format*, *battery_config*)
Bases: `abc.ABC`

A CellType describes a generic lithium_ion cell type. Abstract methods have to be completed in inherited classes, e.g. LFP.

check_soc_range(*soc*)

abstract close()

get_calendar_capacity_loss_start()

get_calendar_resistance_increase_start()

get_capacity(*battery_state*)

Determines the current capacity of the battery.

Attention: depending on the battery topology here the capacity refers possibly to that of single cell, module, pack and so on.

Parameters **battery_state** (LithiumIonState) – Current BatteryState of the lithium-ion battery. Used to determine if the avaiable capacity depending on C-rate and temperature, if applicable for cell type.

Returns battery capacity in Ah

Return type float

get_convection_coefficient()

determines the convective heat transfer coefficient of a battery cell

Returns convective heat transfer coefficient in W/(m^2*K)

Return type float

get_coulomb_efficiency(*battery_state*)

Determines the coulomb efficiency based on if the lithium_ion is charging or discharging.

Default: 1.0 -> losses are calculated via internal resistance

Parameters **battery_state** (LithiumIonState) – Current BatteryState of the lithium_ion. Used to determine if the lithium_ion is charging or discharging.

Returns Coulomb efficiency value.

Return type float

get_cyclic_capacity_loss_start()

get_cyclic_resistance_increase_start()

abstract get_internal_resistance(*battery_state*)

Determines the internal resistance based on the current BatteryState and a lookup table for every cell type.

Parameters **battery_state** (LithiumIonState) – Current state of the lithium_ion.

Returns Internal resistance of the cell in Ohm

Return type float

get_mass()
determines the mass of battery cell

Returns mass in kg

Return type float

get_max_current(*battery_state*)
determines the maximum operation current of a battery cell Attention: the sign of current is defined in the convention that charging current is positive and discharging is negative, therefore the minimal value is defined with regard to the sign.

Parameters **battery_state** (LithiumIonState) – Current BatteryState of the lithium_ion. Used to determine if the lithium_ion is charging or discharging and to get the SOC and temperature

Returns maximum current in A

Return type float

get_max_temp()
determines the maximum allowed operation temperature of a given cell type

Returns maximum operation temperature in K

Return type float

get_max_voltage()
determines the recommended maximal operation voltage of a battery cell according to data sheet

Returns maximum operation voltage in V

Return type float

get_min_current(*battery_state*)
determines the minimal operation current of a battery cell Attention: the sign of current is defined in the convention that charging current is positive and discharging is negative, therefore the minimal value is defined with regard to the sign.

Parameters **battery_state** (LithiumIonState) – Current BatteryState of the lithium_ion. Used to determine if the lithium_ion is charging or discharging and to get the SOC and temperature

Returns minimal current in A

Return type float

get_min_temp()
determines the minimal allowed operation temperature of a given cell type

Returns minimal operation temperature in K

Return type float

get_min_voltage()
determines the recommended minimal operation voltage of a battery cell according to data sheet

Returns minimal operation voltage in V

Return type float

get_name()
determines the class name of a cell (E.g. SonyLFP)

Returns class name of a cell

Return type str

get_nominal_capacity()
Determines the capacity of the battery under nominal conditions.

Attention: depending on the battery topology here the capacity refers possibly to that of single cell, module, pack and so on.

Returns battery capacity in Ah

Return type float

get_nominal_voltage()
determines the nominal voltage of the given cell type.

Returns Nominal voltage in V

Return type float

abstract get_open_circuit_voltage(*battery_state*)
Determines the open circuit voltage based on the current BatteryState and a lookup table for every cell type.

Parameters **battery_state** (`LithiumIonState`) – Current state of the lithium_ion.

Returns Open circuit voltage of the cell in V

Return type float

get_parallel_scale()
determines the number of parallelly connected battery cells

Returns number of parallelly connected cells

Return type float

get_resistance_increase_start()

get_self_discharge_rate(*battery_state*)
determines the self-discharge rate of battery cell

Returns self-discharge rate in Wh/s

Return type float

get_serial_scale()
determines the number of serially connected battery cells

Returns number of serially connected cells

Return type float

get_soh_start()

get_specific_heat()
determines the specific heat capacity of a battery cell

Returns specific heat capacity in J/(kg*K)

Return type float

get_surface_area()
determines the surface area of battery cell depending on the cell geometry

Returns cell surface area in m^2

Return type float

get_volume()

Module contents

simses.technology.lithium_ion.degradation package

Subpackages

simses.technology.lithium_ion.degradation.calendar package

Submodules

class CalendarDegradationModel(*cell_type*)
Bases: `abc.ABC`

Degradation Model for the calendaric aging of the ESS.

abstract calculate_degradation(*time*, *battery_state*)
update the calendric capacity loss of a battery Attention: without considering internal aging variation the capacity loss calculation is conducted on a down-scaled single cell and again up-scaled to module or pack assuming that each single cell has the same degradation state :param battery_state: :type battery_state: LithiumIonState :param Current BatteryState of the lithium_ion. Used to determine if the lithium_ion is charging or discharging.:

abstract calculate_resistance_increase(*time*, *battery_state*)
update the calendric internal resistance increase of a battery Attention: without considering internal aging variation the resistance increase calculation is conducted on a down-scaled single cell and again up-scaled to module or pack assuming that each

single cell has the same degradation state :param battery_state: :type battery_state: LithiumIonState :param Current BatteryState of the lithium_ion. Used to determine if the lithium_ion is charging or discharging.:

abstract close()

Closing all resources in calendar degradation model

abstract get_degradation()

get the updated calendric capacity loss caused by the current calculation step (differential capacity loss)

Returns differential capacity loss of the current step in [Ah]

Return type float

abstract get_resistance_increase()

get the updated calendric resistance increase til the current step (differential increase) :returns: differential resistance increase til the current step in [p.u.] :rtype: float

abstract reset(*battery_state*)

resets all values within a calendar degradation model, if battery is replaced :param battery_state: :type battery_state: LithiumIonState; Current BatteryState of the lithium_ion.

class GenericCellCalendarDegradationModel(*cell_type*, *battery_config*)

Bases: *simses.technology.lithium_ion.degradation.calendar.calendar_degradation.CalendarDegradationModel*

calculate_degradation(*time*, *battery_state*)

calculate_resistance_increase(*time*, *battery_state*)

close()

get_degradation()

get_resistance_increase()

reset(*battery_state*)

class SonyLFPCalendarDegradationModel(*cell_type*, *battery_data_config*, *battery_config*)

Bases: *simses.technology.lithium_ion.degradation.calendar.calendar_degradation.CalendarDegradationModel*

calculate_degradation(*time*, *battery_state*)

calculate_resistance_increase(*time*, *battery_state*)

close()

get_degradation()

get_resistance_increase()

reset(*battery_state*)

class PanasonicNCACalendarDegradationModel(*cell_type*)

Bases: *simses.technology.lithium_ion.degradation.calendar.calendar_degradation.CalendarDegradationModel*

calculate_degradation(*time*, *battery_state*)

calculate_resistance_increase(*time*, *battery_state*)

close()

get_degradation()

get_resistance_increase()

reset(*battery_state*)

class MolicelNMCCalendarDegradationModel(*cell_type*, *battery_data_config*)

Bases: *simses.technology.lithium_ion.degradation.calendar.calendar_degradation.CalendarDegradationModel*

calculate_degradation(*time*, *battery_state*)

calculate_resistance_increase(*time*, *battery_state*)

close()

get_degradation()

get_resistance_increase()

get_stressfkt_ca_cal(*battery_state*)

get the stress factor for calendar aging capacity loss

Parameters battery_state (*state including soc and temperature*) –

Returns float

Return type stress parameters of calendar aging (capacity loss)

get_stressfkt_ri_cal(*battery_state*)

get the stress factor for calendar aging resistance increase

Parameters battery_state (*state including soc and temperature*) –

Returns float

Return type stress parameters of calendar aging (resistance increase)

reset(*battery_state*)

class SanyoNMCCalendarDegradationModel(*cell_type*)

Bases: *simses.technology.lithium_ion.degradation.calendar.calendar_degradation.CalendarDegradationModel*

Source: Schmalstieg, J., Käbitz, S., Ecker, M., & Sauer, D. U. (2014). A holistic aging model for Li (NiMnCo) O2 based 18650 lithium-ion batteries. Journal of Power Sources, 257, 325-334.

calculate_degradation (*time*, *battery_state*)

calculate_resistance_increase (*time*, *battery_state*)

close ()

get_degradation ()

get_resistance_increase ()

reset (*battery_state*)

class NoCalendarDegradationModel

Bases: `simses.technology.lithium_ion.degradation.calendar.calendar_degradation.CalendarDegradationModel`

calculate_degradation (*time*, *battery_state*)

calculate_resistance_increase (*time*, *battery_state*)

close ()

get_degradation ()

get_resistance_increase ()

reset (*battery_state*)

Module contents

simses.technology.lithium_ion.degradation.cyclic package

Submodules

class CyclicDegradationModel (*cell_type*, *cycle_detector*)

Bases: `abc.ABC`

abstract calculate_degradation (*battery_state*)

update the cyclic capacity loss of a battery Attention: without considering internal aging variation the capacity loss calculation is conducted on a down-scaled single cell and again up-scaled to module or pack assuming that each single cell has the same degradation state :param battery_state: :type battery_state: LithiumIonState :param Current BatteryState of the lithium_ion. Used to determine if the lithium_ion is charging or discharging.:

abstract calculate_resistance_increase (*battery_state*)

update the cyclic internal resistance increase of a battery Attention: without considering internal aging variation the resistance increase calculation is conducted on a down-scaled single cell and again up-scaled to module or pack assuming that each single cell has the same degradation state :param battery_state: :type battery_state: LithiumIonState :param Current BatteryState of the lithium_ion. Used to determine if the lithium_ion is charging or discharging.:

abstract close()
Closing all resources in calendar degradation model

abstract get_degradation()
get the updated cyclic capacity loss til the current recognized half cycle (differential capacity loss)

Returns differential cyclic capacity loss til the current recognized half cycle in [Ah]

Return type float

abstract get_resistance_increase()
get the updated cyclic resistance increase caused the current recognized half cycle (differential increase) :returns: differential resistance increase caused by the current step in [p.u.] :rtype: float

abstract reset(*lithium_ion_state*)
resets all values within a cyclic degradation model, if battery is replaced

Parameters **lithium_ion_state** (*LithiumIonState; current State of the lithium_ion*) –

class GenericCellCyclicDegradationModel(*cell_type*, *cycle_detector*, *battery_config*)
Bases: *simses.technology.lithium_ion.degradation.cyclic.cyclic_degradation.CyclicDegradationModel*

calculate_degradation(*battery_state*)

calculate_resistance_increase(*battery_state*)

close()

get_degradation()

get_resistance_increase()

reset(*lithium_ion_state*)

class SonyLFPCyclicDegradationModel(*cell_type*, *cycle_detector*, *battery_data_config*, *battery_config*)
Bases: *simses.technology.lithium_ion.degradation.cyclic.cyclic_degradation.CyclicDegradationModel*

calculate_degradation(*battery_state*)

calculate_resistance_increase(*battery_state*)

close()

get_degradation()

get_resistance_increase()

reset(*lithium_ion_state*)

class PanasonicNCACyclicDegradationModel(*cell_type*, *cycle_detector*)
Bases: `simses.technology.lithium_ion.degradation.cyclic.cyclic_degradation.CyclicDegradationModel`

calculate_degradation(*battery_state*)

calculate_resistance_increase(*battery_state*)

close()

get_degradation()

get_resistance_increase()

reset(*lithium_ion_state*)

class MolicelNMCCyclicDegradationModel(*cell_type*, *cycle_detector*, *battery_data_config*)
Bases: `simses.technology.lithium_ion.degradation.cyclic.cyclic_degradation.CyclicDegradationModel`

calculate_degradation(*battery_state*)

calculate_resistance_increase(*battery_state*)

close()

get_degradation()

get_resistance_increase()

get_stressfkt_ca_cyc(*doc*)
get the stress factor for cyclic aging capacity loss

Parameters **doc** (`depth of cycle`) –

Returns **float**

Return type stress parameters of cyclic aging (capacity loss)

get_stressfkt_ri_cyc(*doc*)
get the stress factor for cyclic aging resistance increase

Parameters **doc** (`depth of cycle`) –

Returns **float**

Return type stress parameters of cyclic aging (resistance increase)

reset(*lithium_ion_state*)

class SanyoNMCCyclicDegradationModel(*cell_type*, *cycle_detector*)
Bases: `simses.technology.lithium_ion.degradation.cyclic.cyclic_degradation.CyclicDegradationModel`

Source: Schmalstieg, J., Käbitz, S., Ecker, M., & Sauer, D. U. (2014). A holistic aging model for Li (NiMnCo) O2 based 18650 lithium-ion batteries. Journal of Power Sources, 257, 325-334.

calculate_degradation(*battery_state*)

calculate_resistance_increase(*battery_state*)

close()

get_degradation()

get_resistance_increase()

reset(*lithium_ion_state*)

class NoCyclicDegradationModel

Bases: `simses.technology.lithium_ion.degradation.cyclic.cyclic_degradation.CyclicDegradationModel`

calculate_degradation(*battery_state*)

calculate_resistance_increase(*battery_state*)

close()

get_degradation()

get_resistance_increase()

reset(*lithium_ion_state*)

Module contents

Submodules

class DegradationModel(*cell*, *cyclic_degradation_model*, *calendar_degradation_model*, *cycle_detector*, *battery_config*, *initial_degradation_possible=False*)

Bases: `abc.ABC`

Model for the degradation behavior of the ESS by analysing the resistance increase and capacity decrease.

calculate_degradation(*time*, *battery_state*)

Calculates the resistance increase and capacity decrease (calendar always and cyclic only, if a cycle was detected)

Parameters

- **time** (`float`) – Current timestamp.
- **battery_state** (`LithiumIonState`) – Current state of the lithium_ion.

check_battery_replacement(*battery_state*)

Checks eol criteria and replaces the battery has to be replaced if necessary

Parameters **battery_state** (`LithiumIonState`) – Current state of the lithium_ion.

close()
Closing all resources in degradation model

update(*time*, *battery_state*)
Updating the capacity and resistance of the lithium_ion through the degradation model.

Parameters

- **time** (*float*) – Current timestamp.
- **battery_state** (LithiumIonState) – Current state of the lithium_ion.

class GenericCellDegradationModel(*cell_type*, *cycle_detector*, *battery_config*)
Bases: *simses.technology.lithium_ion.degradation.degradation_model.DegradationModel*

class SonyLFPDegradationModel(*cell_type*, *cycle_detector*, *battery_config*, *battery_data_config*)
Bases: *simses.technology.lithium_ion.degradation.degradation_model.DegradationModel*

class DaimlerLMODegradationModel(*cell_type*, *cycle_detector*, *battery_config*)
Bases: *simses.technology.lithium_ion.degradation.degradation_model.DegradationModel*

Degradation Model for Second-Life cell. Specific model unknown, therefore using different existing degradation models from SimSES.

class PanasonicNCADegradationModel(*cell_type*, *cycle_detector*, *battery_config*)
Bases: *simses.technology.lithium_ion.degradation.degradation_model.DegradationModel*

class MolicelNMCDegradationModel(*cell_type*, *cycle_detector*, *battery_config*, *battery_data_config*)
Bases: *simses.technology.lithium_ion.degradation.degradation_model.DegradationModel*

class Samsung94AhNMCDegradationModel(*cell_type*, *cycle_detector*, *battery_config*)
Bases: *simses.technology.lithium_ion.degradation.degradation_model.DegradationModel*

Degradation Model for Samsung94Ah cell. Specific model unknown, therefore using different existing degradation models from SimSES.

class SanyoNMCDegradationModel(*cell_type*, *cycle_detector*, *battery_config*)
Bases: *simses.technology.lithium_ion.degradation.degradation_model.DegradationModel*

class NoDegradationModel(*cell*, *cycle_detector*, *battery_config*)

Bases: *simses.technology.lithium_ion.degradation.degradation_model.DegradationModel*

Module contents

simses.technology.lithium_ion.equivalent_circuit_model package

Submodules

class EquivalentCircuitModel
Bases: `abc.ABC`

abstract close()
Closing all resources in lithium_ion model

abstract update(*time*, *battery_state*)
Updating current, voltage, power loss and soc of lithium_ion state

class RintModel(*cell_type*)
Bases: *simses.technology.lithium_ion.equivalent_circuit_model.equivalent_circuit.EquivalentCircuitModel*

close()

update(*time*, *battery_state*)

Module contents

simses.technology.lithium_ion.test package

Submodules

create_battery_state()

test_current(*uut*, *current*, *result*)

test_fulfillment(*uut*, *current*, *result*)

test_soc(*uut*, *soc*, *current*, *result*)

test_temperature(*uut*, *temperature*, *current*, *result*)

uut()

class TestCellType
Bases: `object`

lithium_ion_state: simses.commons.state.technology.lithium_ion.Lithi

lithium_ion_subclass_list()

make_lithium_ion_cell(*lithium_ion_subclass*)

step = 10

test_ocv (*soc*, *lithium_ion_subclass_list*)

test_values_soc = array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,

Module contents

Submodules

class LithiumIonBattery (*cell*, *voltage*, *capacity*, *soc*, *soh*, *data_export*, *temperature*, *storage_id*, *system_id*, *config*)

Bases: `simses.technology.storage.StorageTechnology`

Battery orchestrates its models for lithium_ion management system, degradation and thermal management as well as its equivalent circuit

close ()

Closing all resources in lithium_ion

property convection_coefficient

distribute_and_run (*time*, *current*, *voltage*)

get_auxiliaries ()

get_equilibrium_state_for (*time*, *current*, *fixed_values=False*)

Starting update of lithium_ion

get_system_parameters ()

property id

property mass

property max_current

property max_voltage

property min_current

property min_voltage

property specific_heat

property state

property surface_area

property volume

wait ()

class BatteryCircuit (*batteries*)

Bases: `simses.technology.storage.StorageTechnology`

BatteryCircuit is tasked to directly connect several battery types. It handles the balancing currents between the batteries by equalizing the voltage difference.

TODO

1) Having static max / min values for current and voltage or dynamic for each battery type and state?

-> Not needed anymore

2) How to calculate the equilibrium balancing current between batteries?

-> Calculate balancing current depending on OCV an internal resistence of each battery

3) **How to avoid divergency for specific LIB types? Are LIB types correctly modelled? Why do**
   Working: PanasonicNCA, MolicelNMC, SanyoNMC, AkasolAkmNMC; Not working: SonyLFP, GenericCell, AkasolOemNMC;

-> All battery types should work

4) Current alogrithm works better with higher capacities / lower C-Rates

5) Right now the overall current is split between the batteries depending on their capacities. One way to a faster voltage convergence could be to only discharge the battery with higher voltage and charge only the battery with a lower voltage with the applied current.

-> Current is split depending on OCV and internal resistance, in the same function which also calculate the balance currents

6) During the balancing process batteries partly cannot handle the balancing current, hence it takes time for converging - in standard simulation around 30 min.

-> If batteries can not handle the balancing current or overall current (BMS is limiting), the BMS
If this happens, the scenario is not feasable.

close()
Closing all resources in lithium_ion circuit

property convection_coefficient

distribute_and_run(*time*, *current*, *voltage*)

get_auxiliaries()

get_system_parameters()

property mass

property specific_heat

property state

property surface_area

property volume

wait()

class LithiumIonFactory(*config*)
Bases: `object`

close()

create_battery_management_system_from(*cell_type*, *battery_management_system=None*)

create_battery_model_from(*cell_type*, *battery_model=None*)

create_battery_state_from(*system_id*, *storage_id*, *cell_type*, *temperature*, *soc*)

create_cell_type(*cell_type*, *voltage*, *capacity*, *soh*)

create_cycle_detector(*start_soc*)

create_degradation_model_from(*cell_type*, *battery_state*)

Module contents

simses.technology.redox_flow package

Subpackages

simses.technology.redox_flow.degradation package

Submodules

class CapacityDegradationModel(*capacity*)

Bases: `abc.ABC`

Model for the capacity degradation effects of a redox flow battery, which can not be reversed by electrolyte remixing.

abstract close()

Closing all resources in CapacityDegradationModel.

abstract get_capacity_degradation(*time*, *redox_flow_state*)

Determination of the capacity degradation.

Parameters

- **time** (*float*) – Current simulation time in s.
- **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.

Returns Capacity degradation per time step in Wh.

Return type float

update(*time*, *redox_flow_state*)

Calculates the capacity degradation and updates the value as well as the state-of-health (SOH).

Parameters

- **time** (*float*) – Current simulation time in s.
- **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.

class ConstHydrogenCurrent(*capacity*, *stack_module*)

Bases: *simses.technology.redox_flow.degradation.capacity_degradation.CapacityDegradationModel*

Simplified model that considers the reduction in capacity due to a constant hydrogen current for a redox flow battery.

close()

get_capacity_degradation(*time*, *redox_flow_state*)

class NoDegradation(*capacity*)

Bases: *simses.technology.redox_flow.degradation.capacity_degradation.CapacityDegradationModel*

Model with no capacity degradation for a redox flow battery.

close()

get_capacity_degradation(*time*, *redox_flow_state*)

class VariableHydrogenCurrent(*capacity*, *stack_module*, *redox_flow_data_config*)

Bases: *simses.technology.redox_flow.degradation.capacity_degradation.CapacityDegradationModel*

Model that considers the reduction in capacity due to a hydrogen current for a redox flow battery. The hydrogen current is dependent on the state-of-charge.

close()

get_capacity_degradation(*time*, *redox_flow_state*)

Module contents

simses.technology.redox_flow.electrochemical package

Subpackages

simses.technology.redox_flow.electrochemical.control package

Submodules

class RedoxControlSystem(*stack_module*, *electrolyte_system*, *pump_algorithm*, *redox_flow_config*)

Bases: object

RedoxControlSystem class for redox flow batteries. It contains queries to check whether the operation conditions are met.

battery_fulfillment_calc(*redox_flow_state*)
Calculates the battery fulfillment [0, 1].

Parameters **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.

check_current_in_range(*redox_flow_state*, *time*)
Checks if the current is in range. The maximal and minimal current are defined to have enough reactants at the current flow rate or in the stack electrolyte volume.

Parameters

- **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.
- **time** (*float*) – Current simulation time in s.

Returns True if current is in range.

Return type bool

check_soc_in_range(*redox_flow_state*, *delta_soc*, *delta_soc_stack*)
Checks if the state-of-charge (SOC) is to high or to low for charging or discharging.

Parameters

- **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.
- **delta_soc** (*float*) – State-of-charge in p.u.
- **delta_soc_stack** (*float*) – State-of-charge in the stack in p.u.

Returns True if SOC of the system and in the stack is in range for charging or discharging.

Return type bool

check_temperature(*redox_flow_state*)
Checks if the temperature of the electrolyte is within the operation range.

Parameters **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.

Returns True if the temperature is within the operation range.

Return type bool

check_voltage_in_range(*redox_flow_state*)
Checks if voltage is in range between maximal and minimal stack module voltage.

Parameters **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.

Returns True if voltage is between max. and min. stack module voltage.

Return type bool

close()

Closing all resources in RedoxControlSystem.

get_max_current(*redox_flow_state*, *time*)

Determines the maximal faraday current of a stack module (maximal charge current) based on the flow rate. If the flow rate is 0, the electrolyte volume in the stack modules and the still usable concentration of reactants is used for calculation.

Parameters

- **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.
- **time** (*float*) – Current simulation time in s.

Returns Maximal faraday current (=max charge current) in A.

Return type float

get_min_current(*redox_flow_state*, *time*)

Determines the minimal faraday current of a stack module (maximal discharge current) based on the flow rate. If the flow rate is 0, the electrolyte volume in the stack modules and the still usable concentration of reactants is used for calculation.

Parameters

- **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.
- **time** (*float*) – Current simulation time in s.

Returns Minimal faraday current (=max discharge current) in A.

Return type float

set_max_power(*redox_flow_state*)

voltage_in_range(*redox_flow_state*)

If the voltage is outside of the maximal or minimal stack module voltage, then it is set to the maximal or minimal value.

Parameters **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.

Returns Voltage in V.

Return type float

Module contents

Submodules

class ElectrochemicalModel
Bases: `abc.ABC`

Model that calculates the current and voltage of the redox flow stack module.

abstract close()
Closing all resources in ElectrochemicalModel.

abstract update(*time*, *redox_flow_state*)
Updating power (if changes due to redox flow management system), current, voltage, power loss and soc of redox_flow_state. In the update function the control system requests are implemented.

Parameters

- **time** (*float*) – Current simulation time in s.
- **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.

Returns

Return type None

class RintModel(*stack_module*, *control_system*, *electrolyte_system*, *pump_algorithm*)
Bases: `simses.technology.redox_flow.electrochemical.abstract_electrochemical.ElectrochemicalModel`

Equivalent circuit model that calculates the current and voltage of the redox flow stack module based on an internal resistance.

close()

update(*time*, *redox_flow_state*)

Module contents

simses.technology.redox_flow.pump_algorithm package

Submodules

class PumpAlgorithm(*pump*)
Bases: `abc.ABC`

abstract close()
Closing all resources in PumpAlgorithm.

abstract get_flow_rate_anolyte(*redox_flow_state*)
Determines the the flow rate of the anolyte side in m^3/s.

Parameters redox_flow_state (RedoxFlowState) – Current state of redox_flow.

Returns Flow rate of the anolyte side in m^3/s.

Return type float

abstract get_flow_rate_catholyte(*redox_flow_state*)
Determines the the flow rate of the catholyte side in m^3/s.

Parameters redox_flow_state (RedoxFlowState) – Current state of redox_flow.

Returns Flow rate of the catholyte side in m^3/s.

Return type float

abstract get_flow_rate_max()
Maximal needed flow rate in the system in m^3/s.

Returns Maximal flow rate in m^3/s.

Return type float

abstract get_flow_rate_min()
Minimal needed flow rate in the system in m^3/s.

Returns Minimal flow rate in m^3/s.

Return type float

abstract get_pressure_drop_anolyte(*redox_flow_state*)
Determines the pressure drop over the stack module and pipe system for the anolyte side in Pa.

Parameters redox_flow_state (RedoxFlowState) – Current state of redox_flow.

Returns Pressure drop of the anolyte side in Pa.

Return type float

abstract get_pressure_drop_catholyte(*redox_flow_state*)
Determines the pressure drop over the stack module and pipe system for the catholyte side in Pa.

Parameters redox_flow_state (RedoxFlowState) – Current state of redox_flow.

Returns Pressure drop of the catholyte side in Pa.

Return type float

get_soc_begin()

update (*redox_flow_state*, *soc_start_time_step*)
Updates flow rate and pressure drop of the current redox_flow_state and starts the calculation of the pump_power.

Parameters

- **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.
- **soc_start_time_step** (*float*) – SOC at the begin of the time step in p.u.

class ConstantPressurePulsed (*stack_module*, *pump*, *electrolyte_system*, *time_step*)
Bases: `simses.technology.redox_flow.pump_algorithm.abstract_pump_algorithm.PumpAlgorithm`

Pump algorithm with a fix pressure drop that pulses the discharge and charge flow rate for a certain time period when the voltage is drops below the minimal voltage (discharging) or rises above the maximal voltage (charging).

close ()

get_flow_rate_anolyte (*redox_flow_state*)

get_flow_rate_catholyte (*redox_flow_state*)

get_flow_rate_max ()

get_flow_rate_min ()

get_pressure_drop_anolyte (*redox_flow_state*)

get_pressure_drop_catholyte (*redox_flow_state*)

class FixFlowRateStartStop (*stack_module*, *pump*, *electrolyte_system*)
Bases: `simses.technology.redox_flow.pump_algorithm.abstract_pump_algorithm.PumpAlgorithm`

Pump algorithm with a fixed flow rate. The value is area specific.

close ()

get_flow_rate_anolyte (*redox_flow_state*)

get_flow_rate_catholyte (*redox_flow_state*)

get_flow_rate_max ()

get_flow_rate_min ()

get_pressure_drop_anolyte (*redox_flow_state*)

get_pressure_drop_catholyte (*redox_flow_state*)

class StoichFlowRate (*stack_module*, *pump*, *electrolyte_system*, *redox_flow_config*)
Bases: `simses.technology.redox_flow.pump_algorithm.abstract_pump_algorithm.PumpAlgorithm`

Pump algorithm with constant stoichimetric factor. The flow rate is calculated using the stoichimetric factor and the maximal still usable state-of-charge (SOC) difference for charging or discharging.

close()

get_flow_rate(*redox_flow_state*)
Calculates the flow rate based on an stoichiometric factor.

Parameters **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.

Returns Flow rate in m^3/s.

Return type float

get_flow_rate_anolyte(*redox_flow_state*)

get_flow_rate_catholyte(*redox_flow_state*)

get_flow_rate_max()

get_flow_rate_min()

get_pressure_drop_anolyte(*redox_flow_state*)

get_pressure_drop_catholyte(*redox_flow_state*)

Module contents

simses.technology.redox_flow.stack package

Subpackages

simses.technology.redox_flow.stack.electrolyte package

Submodules

class ElectrolyteSystem(*capacity*)
Bases: `abc.ABC`

The ElectrolyteSystem describes the properties of the liquid storage medium of the redox flow battery.

abstract close()
Closing all resources in ElectrolyteSystem.

abstract get_capacity_density()
Returns the volume specific capacity density of the electrolyte in As/m^3.

Returns Volume specific capacity density in As/m^3.

Return type float

get_electrolyte_density()
Returns the mean density of the electrolyte in kg/m^3.

Returns Density of the electrolyte in kg/m^3.

Return type float

abstract get_max_temperature()
Returns the maximal temperature of the electrolyte.

Returns Maximal electrolyte temperature in K.

Return type float

abstract get_max_viscosity()
Determines the maximal viscosity during operation of the redox flow battery.

Returns Maximal viscosity in Pas.

Return type float

abstract get_min_temperature()
Returns the minimal temperature of the electrolyte.

Returns Minimal electrolyte temperature in K.

Return type float

abstract get_min_viscosity()
Determines the minimal viscosity during operation of the redox flow battery.

Returns Minimal viscosity in Pas

Return type float

get_start_capacity()
Returns the total start capacity of the electrolyte of the redox flow system.

Returns Start capacity in Wh.

Return type float

abstract get_vanadium_concentration()
Returns the total vanadium concentration of the electrolyte.

Returns total vanadium concentration of the electrolyte in mol/m^3.

Return type float

abstract get_viscosity_anolyte(*redox_flow_state*)
Determines the anolyte viscosity depending on state-of-charge (SOC) and temperature.

Parameters **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.

Returns Analyte viscosity in Pas.

Return type float

abstract get_viscosity_catholyte(*redox_flow_state*)

Determines the anolyte viscosity depending on the state-of-charge (SOC) and temperature.

Parameters **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.

Returns Catholyte viscosity in Pas.

Return type float

class VanadiumSystem(*capacity*, *redox_flow_config*)

Bases: *simses.technology.redox_flow.stack.electrolyte.abstract_electrolyte.ElectrolyteSystem*

The parameters of VanadiumSystem are based on experimental data of an electrolyte consisting of 1.6 M Vanadium in aqueous sulphuric acid (2 M H2SO4) from GfE (Gesellschaft für Elektrometallurgie mbH).

MAX_ELECTROLYTE_TEMPERATURE = 313.15

MIN_ELECTROLYTE_TEMPERATURE = 283.15

close()

get_capacity_density()

get_electrolyte_density()

get_max_temperature()

get_max_viscosity()

get_min_temperature()

get_min_viscosity()

get_vanadium_concentration()

get_viscosity_anolyte(*redox_flow_state*)

The parameter for the viscosity are based on experimental measurements performed at ZAE Bayern by Lisa Hoffmann. Literature source: Hoffmann, Lisa. Physical properties of a VRFB-electrolyte and their impact on the cell-performance. master theses. RWTH Aachen, 2018. The temperature dependency at SOC 50 % was extrapolated to the other SOC values.

get_viscosity_catholyte(*redox_flow_state*)

The parameter for the viscosity are based on experimental measurements performed at ZAE Bayern by Lisa Hoffmann. Literature source: Hoffmann, Lisa. Physical properties of a VRFB-electrolyte and their impact on the cell-performance. master theses. RWTH Aachen, 2018. The temperature dependency at SOC 50 % was extrapolated to the other SOC values.

Module contents

Submodules

class StackModule(*electrolyte_system*, *voltage*, *power*, *cell_number_per_stack*, *stack_power*, *redox_flow_config*)

Bases: `abc.ABC`

A StackModule describes a module of electrical serial and parallel connected redox flow stacks.

abstract close()

Closing all resources in StackModule.

dependent_parameters()

Boolean value that signals whether parameters for the electrochemical model are dependent on voltage or current and therefore require iterations (e. q. the internal resistance is dependent on the current).

Returns True if parameters for the electrochemical model are dependent on voltage or current.

Return type bool

abstract get_cell_per_stack()

Determines the cells per stack for the used stack type.

Returns Number of cells per stack.

Return type int

abstract get_electrode_porosity()

Returns the electrode porosity.

Returns Electrode porosity.

Return type float

get_electrode_thickness()

Returns the electrode thickness in m.

Returns Electrode thickness in m.

Return type float

abstract get_hydraulic_resistance()

Returns the hydraulic resistance of the system in 1/m^3. The electrolyte flows through all stacks parallel.

Returns Hydraulic resistance in 1/m^3.

Return type float

abstract get_internal_resistance(*redox_flow_state*)

Determines the internal resistance of the stack module based on the current RedoxFlowState.

Parameters **redox_flow_state** (`RedoxFlowState`) – Current state of redox_flow.

Returns Internal resistance of the stack module in Ohm.

Return type float

get_max_current_high_soc()

Determines the maximal current at the maximal voltage.

Returns Maximal current in A at the maximal voltage.

Return type float

get_max_current_low_soc()

Determines the maximal current of the stack module at the lowest soc.

Returns Maximal current in A.

Return type float

get_max_power()

Returns the maximal power of the redox flow system.

Returns Maximal Power of the redox flow system in W.

Return type float

abstract get_max_voltage()

Determines the maximal voltage of a stack module.

Returns Maximal stack module voltage in V.

Return type float

abstract get_min_voltage()

Determines the minimal voltage of a stack module.

Returns Minimal stack module voltage in V.

Return type float

get_name()

Determines the class name of a stack typ (e. g. CellDataStack5500W).

Returns Class name of a stack typ.

Return type str

abstract get_nominal_voltage_cell()

Returns the nominal voltage of a single cell of a stack module in V. The value is used to change the capacity in Ws to its value in As and vice versa. The value of the OCV at SOC = 50 % and temperature = 30 °C is used.

Returns Nominal cell voltage in V.

Return type float

abstract get_open_circuit_voltage(*redox_flow_state*)

Calculates the open circuit voltage (OCV) of a stack module depended on the electrolyte.

Parameters **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.

Returns OCV of the stack module in V.

Return type float

get_parallel_scale()

Returns the parallel scale of stacks in the stack module. The value can be float if exact_size is true.

Returns Number of parallel stacks in the stack module.

Return type float

abstract get_self_discharge_current(*redox_flow_state*)

Determines the self-discharge current that discharges the stack module during standby and which accounts for self-discharge losses during cycling (Coulomb efficiency). The connection of multiple cells and stacks is considered.

Parameters **redox_flow_state** (RedoxFlowState) – Current state of redox_flow.

Returns Total self-discharge current for a stack module in A.

Return type float

get_serial_scale()

Returns the serial scale of stacks in the stack module. The value can be float if exact_size is true.

Returns Number of serial stacks in the stack module.

Return type float

abstract get_specific_cell_area()

Returns the specific geometrical electrode area in cm^2. This corresponds to the area of the electrode that presses on the membrane.

Returns Cell area in cm^2.

Return type float

get_stacks_electrolyte_volume()

Returns the volume of electrolyte in the stack module electrodes in m^3 for one electrolyte side.

Returns Electrolyte volume in the electrodes of the stack module in m^3.

Return type float

get_total_electrolyte_volume()

Returns the total volume of the electrolyte of on electrolyte side (anolyte or catholyte).

Returns Electrolyte volume of one side in m^3.

Return type float

class CellDataStack5500W(*electrolyte_system*, *voltage*, *power*, *redox_flow_data_config*, *redox_flow_config*)

Bases: `simses.technology.redox_flow.stack.abstract_stack.StackModule`

CellDataStack5500W is a stack based on experimental data of single cell measurements and scaled up to a 40-cell stack.

close()

dependent_parameters()

get_cell_per_stack()

get_electrode_porosity()

get_electrode_thickness()

get_hydraulic_resistance()

get_internal_resistance(*redox_flow_state*)

Experimental data based on single cell measurements at ZAE Bayern. Source: SimSES: Möller, Marc et al. A holistic simulation framework for modelingand analyzing stationary energy storage systems.

get_max_voltage()

get_min_voltage()

get_nominal_voltage_cell()

Calculated for a temperature of 25 °C and at SOC 50 %.

get_open_circuit_voltage(*redox_flow_state*)

Literature source: Fink, Holger. Untersuchung von Verlustmechanismen in Vanadium-Flussbatterien. Diss. Technische Universität München, 2019. equation 5.18, assumption: SOH = 100 %, therefore ver = 0.5

get_self_discharge_current(*redox_flow_state*)

get_specific_cell_area()

class DummyStack3000W(*electrolyte_system*, *voltage*, *power*, *redox_flow_config*)

Bases: `simses.technology.redox_flow.stack.abstract_stack.StackModule`

DummyStack3000W describes a dummy stack with a constant internal resistance. The dependency of the resistance from SOC, flow rate and current is neglected. This stack type can be used to easily implement other stack parameters.

close()

dependent_parameters()

get_cell_per_stack()

get_electrode_porosity()

get_electrode_thickness()

get_hydraulic_resistance()

get_internal_resistance(*redox_flow_state*)

get_max_voltage()

get_min_voltage()

get_nominal_voltage_cell()
Calculated for a temperature of 25 °C and at SOC 50 %.

get_open_circuit_voltage(*redox_flow_state*)
Literature source: Fink, Holger. Untersuchung von Verlustmechanismen in Vanadium-Flussbatterien. Diss. Technische Universität München, 2019. equation 5.18, assumption: SOH = 100 %, therefore ver = 0.5

get_self_discharge_current(*redox_flow_state*)

get_specific_cell_area()

class DummyStack5500W(*electrolyte_system*, *voltage*, *power*, *redox_flow_config*)
Bases: `simses.technology.redox_flow.stack.abstract_stack.StackModule`

DummyStack5500W describes a dummy stack with a constant internal resistance. The dependency of the resistance from SOC, flow rate and current is neglected. The input data is orientated on the CellDataStack5500W

close()

dependent_parameters()

get_cell_per_stack()

get_electrode_porosity()

get_electrode_thickness()

get_hydraulic_resistance()

get_internal_resistance(*redox_flow_state*)

get_max_voltage()

get_min_voltage()

get_nominal_voltage_cell()
Calculated for a temperature of 25 °C and at SOC 50 %.

get_open_circuit_voltage(*redox_flow_state*)
Literature source: Fink, Holger. Untersuchung von Verlustmechanismen in Vanadium-Flussbatterien. Diss. Technische Universität München, 2019. equation 5.18, assumption: SOH = 100 %, therefore ver = 0.5

get_self_discharge_current(*redox_flow_state*)

get_specific_cell_area()

class IndustrialStack1500W(*electrolyte_system*, *voltage*, *power*, *redox_flow_data_config*, *redox_flow_config*)

Bases: `simses.technology.redox_flow.stack.abstract_stack.StackModule`

IndustrialStack1500W is a stack parameterised with field data of a commercially available redox flow system. The internal resistance considers an increase due to mass transport effects when the pump are off and the power exceeds a certain limit value.

close()

dependent_parameters()

get_cell_per_stack()

get_electrode_porosity()

get_electrode_thickness()

get_hydraulic_resistance()

get_internal_resistance(*redox_flow_state*)

get_max_voltage()

get_min_voltage()

get_nominal_voltage_cell()

Calculated at SOC 50 %.

get_open_circuit_voltage(*redox_flow_state*)

get_self_discharge_current(*redox_flow_state*)

get_specific_cell_area()

class IndustrialStack9000W(*electrolyte_system*, *voltage*, *power*, *redox_flow_data_config*, *redox_flow_config*)

Bases: `simses.technology.redox_flow.stack.abstract_stack.StackModule`

IndustrialStack9000W is a scaled up version of IndustrialStack1500W to achieve a higher stack power. The flow length is kept constant.

close()

dependent_parameters()

get_cell_per_stack()

get_electrode_porosity()

get_electrode_thickness()

get_hydraulic_resistance()

get_internal_resistance(*redox_flow_state*)

get_max_voltage()

get_min_voltage()

get_nominal_voltage_cell()
Calculated at SOC 50 %.

get_open_circuit_voltage(*redox_flow_state*)

get_self_discharge_current(*redox_flow_state*)

get_specific_cell_area()

Module contents

simses.technology.redox_flow.test package

Submodules

class TestClassStackModule
Bases: object

make_stack_module(*stack_module_typ_subclass*)

redox_flow_state: simses.commons.state.technology.redox_flow.RedoxFl

stack_module_subclass_list()

test_current_sign_check(*stack_module_subclass_list*)

test_serial_scale_calculation(*stack_module_subclass_list*)

Module contents

Submodules

class RedoxFlowFactory(*config*)
Bases: object

check_pump_algorithm(*pump_algorithm, stack_module*)

close()

create_degradation_model(*capacity, stack_module*)

create_electrochemical_model(*stack_module, battery_management_system, electrolyte_system, pump_algorithm, electrochemical_model=None*)
Initial creates the ElectrochemicalModel object for a specific model, which includes the battery management system requests.

Parameters

- **stack_module** (`StackModule`) – stack module of a redox flow battery
- **battery_management_system** (`RedoxControlSystem`) – battery management system of the redox flow battery
- **electrolyte_system** (`ElectrolyteSystem`) – electrolyte system of the redox flow battery
- **pump_algorithm** (`PumpAlgorithm`) – pump algorithm ot the redox flow battery
- **electrochemical_model** (`ElectrochemicalModel`) – electrochemical model of the redox flow battery

Returns

Return type *ElectrochemicalModel*

create_electrolyte_system (*capacity*, *stack_type*)
Initial creates the ElectrolyteSystem object.

Parameters

- **capacity** (*float*) – Total start capacity of the redox flow system in Wh.
- **stack_type** (*str*) – stack type of the redox flow battery

Returns

Return type *ElectrolyteSystem*

create_pump_algorithm (*pump*, *stack_module*, *electrolyte_system*, *pump_algorithm*)

create_pumps (*pump_algorithm*)

create_redox_flow_state_from (*storage_id*, *system_id*, *stack_module*, *capacity*, *temperature*, *redox_flow_state=None*)
Initial creates the RedoxFlowState object if it doesn't exist.

Parameters

- **storage_id** (*int*) – storage id
- **system_id** (*int*) – system id
- **stack_module** (`StackModule`) – stack module based on specific stack typ
- **capacity** (*float*) – Capacity of the electrolyte of the redox flow battery in Wh.
- **temperature** (*float*) – Temperature of the electrolyte of the redox flow battery in K.
- **redox_flow_state** (`RedoxFlowState`) –

Returns state of the redox flow battery

Return type *RedoxFlowState*

create_redox_management_system(*stack_module*, *electrolyte_system*, *pump_algorithm*, *redox_management_system=None*)

Initial creates the BatteryManagementSystem object of the redox flow battery.

Parameters

- **stack_module** (`StackModule`) – stack module of a redox flow battery
- **electrolyte_system** (`ElectrolyteSystem`) – management system of a redox flow battery
- **pump_algorithm** (`PumpAlgorithm`) – pump algorithm of the redox flow battery
- **redox_management_system** (`RedoxControlSystem`) – battery management system of the redox flow battery

Returns

Return type *RedoxControlSystem*

create_stack_module(*stack_module*, *electrolyte_system*, *voltage*, *power*)

Initial creates the StackModule object for a specific stack typ.

Parameters

- **stack_module** (*str*) – stack type for stack module
- **electrolyte_system** (`ElectrolyteSystem`) – electrolyte system
- **voltage** (*float*) – nominal stack module voltage in V of the redox flow battery
- **power** (*float*) – nominal stack module power in W of the redox flow battery

Returns

Return type *StackModule*

class RedoxFlow(*stack_type*, *power*, *voltage*, *capacity*, *pump_algorithm*, *temperature*, *data_export*, *storage_id*, *system_id*, *config*)

Bases: `simses.technology.storage.StorageTechnology`

The RedoxFlow class updates the electrochemical model that includes the redox control system, the degradation model and the pump algorithm.

close()

Closing all resources in redox_flow_system

property convection_coefficient

Convective heat transfer coefficient of a redox flow battery.

A value of 5 W/m^2/K is assumed for the convective heat transfer coefficient, which is in the same order of magnitude as the reported values from Tang et al. (2012) and Yan et al. (2016). If the thermal model is to be used for a particular redox flow battery geometry, the value must be adjusted with respect to the system specifications. Tang, Ao, et al. "Thermal modelling and simulation of the all-vanadium redox flow battery." Journal of Power Sources 203 (2012): 165-176. Yan, Yitao, et al. "Modelling and simulation of thermal behaviour of vanadium redox flow battery." Journal of Power Sources 322 (2016): 116-128.

Returns Convective heat transfer coefficient in W/(m^2*K).

Return type float

`distribute_and_run`(*time*, *current*, *voltage*)

`get_auxiliaries`()

`get_equilibrium_state_for`(*time*, *rfbs*, *soc_start_time_step*, *soc_stack_start_time_step*)

`get_system_parameters`()

`property mass`

Mass of the storage technology in kg.

Returns Mass in kg.

Return type float

`set`(*time*, *current*, *voltage*)

Sets the new simulation time an sets the power (current * voltage) of the RedoxFlowState for the next simulation time step.

Parameters

- **`time`** (*`float`*) – current time of the simulation
- **`current`** (*`float`*) – target current in A
- **`voltage`** (*`float`*) – target voltage in V

Returns

Return type None

`property specific_heat`

Specific heat of storage technology in J/(kgK).

Returns Specific heat capacity in J/[kgK).

Return type float

`property state`

`property surface_area`

Surface area of the storage technology that can be impacted by confection m^2.

Returns Surface area of the storage technology in m^2.

Return type float

update()
Starts updating the calculation for the electrochemical model of the redox flow battery, which includes the battery management system requests.

Returns

Return type None

property volume
Volume of storage technology in m^3.

Returns Redox flow volume in m^3.

Return type float

wait()

Module contents

Submodules

class StorageTechnology
Bases: `abc.ABC`

Abstract class for all technologies

abstract close()
Closing all open resources in a data

abstract property convection_coefficient
determines the convective heat transfer coefficient of a battery cell

Returns convective heat transfer coefficient in W/(m^2*K)

Return type float

abstract distribute_and_run(*time*, *current*, *voltage*)
starts the update process of a data

Parameters

- **time** (*current simulation time*) –
- **current** (*requested current for a data*) –
- **voltage** (*dc voltage for a data*) –

abstract get_auxiliaries()

abstract get_system_parameters()
All system parameters inherited by the storage technology

abstract property mass
Mass of storage technology in kg

abstract property specific_heat
Specific heat of storage technology in J/(kgK)

abstract property state
function to get the data state

Returns StorageTechnologyState

Return type specific data state

abstract property surface_area
Surface area of storage technology in m2

abstract property volume
Volume of storage technology in m3

abstract wait()

Module contents

2.6 Data

The commons package includes the data handling.

2.6.1 simses.data package

Subpackages

simses.data.electrolyzer package

Subpackages

simses.data.electrolyzer.evaluation package

Submodules

Module contents

simses.data.electrolyzer.lookuptable package

Subpackages

simses.data.electrolyzer.lookuptable.archive package

Submodules

Module contents

simses.data.electrolyzer.lookuptable.ui_curves package

Subpackages

simses.data.electrolyzer.lookuptable.ui_curves.activation_overpotential_alkaline package

Module contents

simses.data.electrolyzer.lookuptable.ui_curves.general_alkaline package

Module contents

simses.data.electrolyzer.lookuptable.ui_curves.stuart_hri package

Module contents

Module contents

Submodules

calculate_anode_activation_current(*activation_overvoltage*, *temperature*)

calculate_bubble_rate_coverage(*current*, *temperature*)

calculate_cathode_activation_current(*activation_overvoltage*, *temperature*)

display_example(*temperature*, *compare*)

generate_activation_overvoltage_lookup()

Module contents

simses.data.electrolyzer.parameters package

Module contents

simses.data.electrolyzer.regression package

Submodules

Module contents

Module contents

simses.data.energy_management package

Module contents

simses.data.fuel_cell package

Submodules

B1_Calc(*n=2*)
: Calculate B (Constant in the mass transfer term). :param n: number of moles of electrons transferred in the balanced equation occurring in the fuel cell :type n: int :return: B1 as float

Enernst_Calc(*PH2*, *PO2*)
: Calculate Enernst. :param PH2: partial pressure [atm] :type PH2 : float :param PO2: partial Pressure [atm] :type PO2: float :return: Enernst [V} as float

Eta_Act_Calc(*i*, *alpha*, *i_n*)
: Calculate Eta activation. :param alpha: Ladungsübertragungskoeffizient :type alpha: float :param i: cell current [A/cm2] :type i: float :param i_n: interner Zellstrom [A/cm2] :type i_n: float :param i_0: Austauschstromdichte [A/cm2] :type i_0: float :return: Eta activation [V] as float

Eta_Conc_Calc(*i*, *B*, *imax*)
: Calculate Eta concentration. :param i: cell load current [A/cm2] :type i :float :param imax: maximal cell current [A/cm2] :type imax: float :return: Eta concentration [V] as float

Eta_Ohmic_Calc(*R_total*)
: Calculate Eta ohmic. :param i: cell current [A/cm2] :type i: float :param R_total: Gesamtwiderstand einer Zelle [Ohm] :type R_total: float

Loss_Calc(*Eta_Act*, *Eta_Ohmic*, *Eta_Conc*)
: Calculate loss. :param Eta_Act: Eta activation [V] :type Eta_Act : float :param Eta_Ohmic: Eta ohmic [V] :type Eta_Ohmic : float :param Eta_Conc: Eta concentration [V] :type Eta_Conc : float :return: loss [V] as float

PowerStack_Calc(*Power*, *N*)
: Calculate power_stack. :param Power: single cell power [W/cm2] :type Power : float :param N: number of single cells :type N : int :return: power stack [W] as float

Power_Calc(*Vcell*, *i*)
: Calculate power. :param Vcell: Vcell Voltage [V] :type Vcell : float :param i: cell current [A/cm2] :type i : float :return: cell power [W/cm2] as float

T_cell_Calc()
: Calculate T_cell. :param i: cell current [A/cm2] :type i: float :return: T as float

VStack_Calc(*N*, *Vcell*)
Calculate VStack. :param N: number of single cells :type N: int :param Vcell: cell voltage [V} :type Vcell: float :return: VStack [V] as float

Vcell_Calc(*Enernst*, *Loss*)
Calculate cell voltage. :param Enernst: Nernstvoltage [V} :type Enernst : float :param Loss: loss [V] :type Loss : float :return: cell voltage [V] as float

i_0_Calc_from_temperature()

Parameters

- **i_0_ref** (*float*) – Referenz Austauschstromdichte in [A/cm2]
- **i** (*float*) – cell load current [A/cm2]

Returns i_0 Austauschstromdichte in Abhängigkeit der Temperatur in [A/cm2]

Module contents

simses.data.lithium_ion package

Subpackages

simses.data.lithium_ion.cell package

Module contents

Module contents

simses.data.logo package

Module contents

simses.data.power_electronics package

Module contents

simses.data.profile package

Subpackages

simses.data.profile.economic package

Module contents

simses.data.profile.power package

Module contents

simses.data.profile.technical package

Module contents

Module contents

simses.data.pump package

Module contents

simses.data.redox_flow package

Subpackages

simses.data.redox_flow.degradation package

Module contents

simses.data.redox_flow.stack package

Module contents

Module contents

simses.data.thermal package

Module contents

Module contents

2.7 Analysis

Here the simulation analysis can be found.

2.7.1 simses.analysis package

Subpackages

simses.analysis.data package

Submodules

class Data (*config*, *data*)

Bases: `abc.ABC`

Data is abstract class for providing time series values from the state values generated by SimSES. It provides many convenient methods to access simulation data in order to ease calculations in analysis.

property average_power

Mean value of power series in W

property average_soc

Mean soc in p.u.

abstract property capacity

Series of capacity in kWh

property charge_energy

Charge energy in kWh (as positive values)

property charge_energy_per_year

Series of yearly charge energy in kWh (as positive values)

property charge_energy_series

Series of charge energy in kWh (as positive values)

classmethod close ()

property convert_watt_to_kWh

conversion coefficient in order to transform power in W to energy in kWh

Returns conversion coefficient

Return type float

abstract property dc_power

Series of power values in W

property discharge_energy

Discharge energy in kWh (as positive values)

property discharge_energy_per_year

Series of yearly discharge energy in kWh (as positive values)

property discharge_energy_series

Series of discharge energy in kWh (as positive values)

abstract property energy_difference
Energy difference of start and end point in kWh

abstract classmethod get_system_data(*path*, *config*)
Extracts unique systems data from storage data files in path

Parameters

- **path** (*value folder*) –
- **config** (*simulation data_config in value folder*) –

abstract property id
returns: Data id as string :rtype: str

property initial_capacity
First value of capacity series in kWh

property initial_state_of_health
First value of state of health series in p.u.

property max_soc
Maximal soc of series in p.u.

property min_soc
Minimum soc of series in p.u.

abstract property power
Series of power values in W

abstract property soc
Series of soc values in p.u.

static sort_by_id(*data*, *descending=False*)
In-place sorting of list with Data objects by id (ascending)

Parameters

- **descending** – reverse sorting of data list, default: False
- **data** – list of Data objects

abstract property state_of_health
Series of state of health in p.u.

abstract property storage_fulfillment
Percentage of time the system or battery can fulfill the desired power

abstract property time
Time series in s

class ElectrolyzerData(*config*, *data*)
Bases: *simses.analysis.data.abstract_data.Data*

Provides time series data from ElectrolyzerState

property capacity

property convection_heat

property current

property current_density

property dc_power

property energy_difference

property exchange_current_dens_decrease

property exchange_current_dens_decrease_calendar

property exchange_current_dens_decrease_cyclic

classmethod get_system_data (*path*, *config*)

property hydrogen_outflow

property hydrogen_production

property id

property part_pressure_h2

property part_pressure_o2

property power

property power_compressor

property power_gas_drying

property power_pump

property power_water_heating

property pressure_anode

property pressure_cathode

property resistance_increase

property resistance_increase_calendar

property resistance_increase_cyclic

property sat_pressure_h20

property soc

property soh

property state_of_health

property storage_fulfillment

property temperature

property time

property total_amount_h2_kg

property total_amount_h2_nqm

property total_energy_compressor

property total_energy_gas_drying

property total_energy_h2_lhv

property total_energy_heating

property total_energy_pump

property total_energy_reaction

property total_h2_production

class EnergyManagementData (*config*, *data*)

Bases: *simses.analysis.data.abstract_data.Data*

Provides time series data from EnergyManamgentState

property capacity

property dc_power

property energy_difference

property fcr_max_power

classmethod get_system_data (*path*, *config*)

property id

property idm_power

property load_power

property peakshaving_limit

property power

property pv_power

property soc

property state_of_health

property storage_fulfillment

property time

class FuelCellData (*config*, *data*)

Bases: *simses.analysis.data.abstract_data.Data*

Provides time series data from FuelCellState

property capacity

property current

property current_density

property dc_power

property energy_difference

classmethod get_system_data(*path*, *config*)

property hydrogen_use

property id

property power

property pressure_anode

property pressure_cathode

property soc

property state_of_health

property storage_fulfillment

property temperature

property time

property voltage

class HydrogenData(*config*, *data*)

Bases: *simses.analysis.data.abstract_data.Data*

DEPRECATED - Provides time series data from HydrogenState

property capacity

property convection_heat

property current_density_el

property current_el

property dc_power

property energy_difference

property exchange_current_dens_decrease_calendar_el

property exchange_current_dens_decrease_cyclic_el

property exchange_current_dens_decrease_el

property fulfillment

classmethod get_system_data(*path*, *config*)

property hydrogen_outflow

property hydrogen_production

property id

property part_pressure_h2_el

property part_pressure_o2_el

property power

property power_compressor

property power_gas_drying

property power_pump_el

property power_water_heating_el

property pressure_anode_el

property pressure_cathode_el

property ref_voltage_el

property resistance_increase_calendar_el

property resistance_increase_cyclic_el

property resistance_increase_el

property sat_pressure_h20_el

property soc

property soh_el

property state_of_health

property storage_fulfillment

property tank_pressure

property temperature_el

property time

property total_amount_h2_kg

property total_amount_h2_nqm

property total_energy_compressor

property total_energy_gas_drying

property total_energy_h2_lhv

property total_energy_heating

property total_energy_pump_el

property total_energy_reaction

property total_h2_production

class LithiumIonData(*config*, *data*)

Bases: `simses.analysis.data.abstract_data.Data`

Provides time series data from LithiumIonState

property capacity

property capacity_loss_calendar

property capacity_loss_cyclic

property current

property dc_power

property energy_difference

classmethod get_system_data (*path*, *config*)

property id

property power

property resistance

property resistance_increase

property resistance_increase_calendar

property resistance_increase_cyclic

property soc

property soe

property state_of_health

property storage_fulfillment

property temperature

property time

property voltage

class RedoxFlowData (*config*, *data*)

Bases: *simses.analysis.data.abstract_data.Data*

Provides time series data from RedoxFlowState

property capacity

property charge_current_sec

Charge energy in kWh (as positive values)

property charge_difference

property current

property dc_power

property discharge_current_sec

Discharge energy in kWh (as positive values)

property energy_difference

classmethod get_system_data (*path*, *config*)

property id

property mean_open_circuit_voltage

property power

property pump_power

property resistance

property soc

property soc_stack

property state_of_health

property storage_fulfillment

property temperature

property time

class SystemData(*config*, *data*)

Bases: *simses.analysis.data.abstract_data.Data*

Provides time series data from SystemState

property ac_pe_charging_energy

property ac_pe_discharging_energy

property ac_pe_power

property ac_pe_power_charging_energy_series

property ac_pe_power_discharging_energy_series

property ac_power_target

property ambient_temperature

property aux_energy_charging

property aux_power

property capacity

property dc_power

property dc_power_additional

property dc_power_charging_energy

property dc_power_charging_energy_series

property dc_power_discharging_energy

property dc_power_discharging_energy_series

property dc_power_loss

property dc_power_storage

property dc_power_storage_charging_energy

property dc_power_storage_charging_energy_series

property dc_power_storage_discharging_energy

property dc_power_storage_discharging_energy_series

property dc_voltage

property energy_difference

property final_energy_content

classmethod get_system_data (*path*, *config*)

property hvac_thermal_power

property id

property initial_energy_content

property is_top_level_system

property pe_losses

property power

property soc

property solar_thermal_load

property state_of_health

property storage_fulfillment

property storage_technology_loss_energy

property storage_technology_loss_energy_series

property storage_technology_loss_power

property temperature

property time

property total_aux_losses_energy

property total_pe_losses_energy

Module contents

simses.analysis.evaluation package

Subpackages

simses.analysis.evaluation.economic package

Subpackages

simses.analysis.evaluation.economic.revenue_stream package

Submodules

class RevenueStream(*energy_management_data*, *system_data*, *economic_analysis_config*)

Bases: `abc.ABC`

Calculates the cashflow for each project year that is generated by the given RevenueStream. A RevenueStream can be seen as the financial impact of the various applications of a BESS such as peak shaving, self-consumption, frequency containment reserve, etc..

cash_time_series_to_project_years(*cashflows*, *time*)

Converts a cashflow time series for every timestep to a cashflow time series for every project year. A project year is assumed to be 365 days long. A full leap year would therefore be one project year and one day.

Parameters

- **cashflows** – List of float, representing cashflow for each timestep
- **time** – List of float, unix timestamps for each simulation step

Returns List of float, representing cashflow for each project year.

Return type list of float

abstract close()

abstract get_assumptions()

Returns list of assumptions for the given RevenueStream. Assumptions are handled with the EvaluationResults class.

Returns List of EvaluationResults, representing assumptions

Return type list of EvaluationResult

abstract get_cashflow()

Returns non-discounted cashflow for every project year for the given RevenueStream.

Returns List of cashflow for every project year

Return type list of float

abstract get_evaluation_results()

Returns list of EvaluationResults for the given RevenueStream.

Returns List of EvaluationResults

Return type list of EvaluationResult

set_investment_cost(*value*)

class DemandChargeReduction(*energy_management_data*, *system_data*, *economic_analysis_config*)

Bases: *simses.analysis.evaluation.economic.revenue_stream.abstract_revenue_stream.RevenueStream*

class BillingPeriod

Bases: `object`

MONTHLY: str = 'monthly'

OPTIONS: [<class 'str'>] = ['monthly', 'yearly']

YEARLY: str = 'yearly'

close()

get_assumptions()

get_cashflow()

get_evaluation_results()

class ElectricityConsumptionRevenueStream(*energy_management_data*, *system_data*, *economic_analysis_config*)

Bases: *simses.analysis.evaluation.economic.revenue_stream.abstract_revenue_stream.RevenueStream*

” Calculates the yearly costs for electrictiy consumption for electrolyzer

close()

get_assumptions()

get_cashflow()

get_evaluation_results()

class EnergyCostReduction(*energy_management_data*, *system_data*, *economic_analysis_config*)

Bases: *simses.analysis.evaluation.economic.revenue_stream.abstract_revenue_stream.RevenueStream*

close()

get_assumptions()

get_cashflow()

get_evaluation_results()

class FCRRevenue(*energy_management_data*, *system_data*, *economic_analysis_config*, *general_config*, *market_profile_config*)

Bases: *simses.analysis.evaluation.economic.revenue_stream.abstract_revenue_stream.RevenueStream*

close()

get_assumptions()

get_cashflow()

get_evaluation_results()

class IntradayRechargeRevenue(*energy_management_data*, *system_data*, *economic_analysis_config*, *general_config*, *market_profile_config*)

Bases: *simses.analysis.evaluation.economic.revenue_stream.abstract_revenue_stream.RevenueStream*

close()

get_assumptions()

get_cashflow()

get_evaluation_results()

class OperationAndMaintenanceRevenue(*energy_management_data*, *system_data*, *economic_analysis_config*)

Bases: *simses.analysis.evaluation.economic.revenue_stream.abstract_revenue_stream.RevenueStream*

Calculates the yearly costs due to maintenance and operation of the storage technology

close()

get_assumptions()

get_cashflow()

get_evaluation_results()

Module contents

Submodules

class EconomicEvaluation(*system_data*, *economic_analysis_config*, *revenue_streams*, *config*, *storage_system_config*)

Bases: *simses.analysis.evaluation.abstract_evaluation.Evaluation*

Performs an economic evaluation by iterating through the respective RevenueStreams. Calculates
internal rate of return (IRR), net present value (NPV), etc..

close()

evaluate()

plot()

print_results()

Module contents

simses.analysis.evaluation.plotting package

Submodules

class Axis(*data, label, color='#0074D9', linestyle='solid'*)

Bases: `object`

property color

property data

property label

property linestyle

class MatplotPlotting(*title=' ', path=' '*)

Bases: *simses.analysis.evaluation.plotting.plotter.Plotting*

class Linestyle

Bases: `object`

DASHED = 'dashed'

DASH_DOT = 'dashdot'

DOTTED = 'dotted'

SOLID = 'solid'

bar(*yaxis, bars*)

histogram(*xaxis, yaxis*)

lines(*xaxis, yaxis, secondary=[]*)

sankey_diagram(*node_links*)

show()

subplots(*xaxis, yaxis*)

sunburst_diagram(*categories*)

class PlotlyPlotting(*title, path*)

Bases: *simses.analysis.evaluation.plotting.plotter.Plotting*

class Linestyle

Bases: `object`

DASHED: str = 'dash'

DASH_DOT: str = 'dashdot'

DOTTED: str = 'dot'

SOLID: str = 'solid'

bar (*yaxis*, *bars*)

get_figures ()

histogram (*xaxis*, *yaxis*)

layout (*fig*, *xaxis=None*, *yaxis=None*, *hist=False*)

lines (*xaxis*, *yaxis*, *secondary=[]*)

sankey_diagram (*node_links*)

show (*fig*)

static_figure_index = 1

subplots (*xaxis*, *yaxis*)

sunburst_diagram (*parameters*)

class Plotting
Bases: `abc.ABC`

class Color
Bases: `object`

AC_POWER_BLUE = '#0496FF'

ANODE_GREEN = '#4A5240'

BLACK = '#000000'

BLUE = '#0065BD'

BRIGHT_BLUE = '#0096FF'

BRIGHT_GREEN = '#66FF00'

BROWN = '#964B00'

CATHODE_PINK = '#C585B3'

CURRENT_CYAN = '#98C6EA'

CYAN = '#00FFFF'

DARK_SKY_BLUE = '#8CBED6'

DC_POWER_GREEN = '#06D6A0'

GREEN = '#A2AD00'

GREY = '#D3D3D3'

HEAT_ORANGE = '#FFBC42'

MAGENTA = '#F012BE'

POWER_YELLOW = '#FFDC00'

PURPLE = '#800080'

RED = '#FF4136'

RESISTANCE_BLACK = '#000000'

ROYAL_BLUE = '#4169E1'

SOC_BLUE = '#0065BD'

SOH_GREEN = '#A2AD00'

STACK_PURPLE = '#0E103D'

TEMPERATURE_RED = '#AF1B3F'

VIOLET = '#EE82EE'

VOLTAGE_GREEN = '#A2AD00'

WHITE = '#FFFFFF'

YELLOW = '#FFDC00'

alphanumerize(*string*)

Returns a valid alphanumeric string that can be used for a filename.

Parameters **string** – String to be processed.

abstract bar(*yaxis*, *bars*)

Creates a bar plot by adding traces from the passed axes.

Parameters

- **yaxis** – List of y-axes.
- **bars** – Number of bars per figure

static convert_to_html(*figure*)

Returns a string that can be embedded in an html from a passed figure object.

Parameters **figure** – Figure to be converted to a html-readable string.

static format_time(*time_data*)

Converts list of timestamps into a list of datetimes.

Parameters **time_data** – List of timestamps.

get_figures()

Returns the list of figures saved in the instance of the plotting class instance.

abstract histogram(*xaxis*, *yaxis*)

Creates a histogram object by adding traces from the passed axes.

Parameters

- **xaxis** – x-axis.
- **yaxis** – List of y-axes.

abstract lines(*xaxis*, *yaxis*, *secondary=[]*)

Creates a figure object by adding traces from the passed axes.

Parameters

- **yaxis** – List of y-axes.
- **secondary** – List of secondary axes.

abstract sankey_diagram(*node_links*)
Creates a sankey diagram from the passed nodes and links :param node_links: dict containing source nodes, target nodes, and values for links

abstract subplots(*xaxis, yaxis*)
Creates a figure object consisting of subplots from passed axes.

Parameters

- **xaxis** – x-Axis.
- **yaxis** – List of y-axes.

sunburst_diagram(*categories*)
Creates a sunburst diagram for the energetic losses :param categories: dict containing labels, parents, and values :return:

Module contents

simses.analysis.evaluation.technical package

Submodules

class ElectrolyzerTechnicalEvaluation(*data, config, path*)
Bases: *simses.analysis.evaluation.technical.technical_evaluation.TechnicalEvaluation*

current_dens_plotting()

current_plotting()

degradation_plotting()

evaluate()

fulfillment_plotting()

h2_production_efficiency_lhv()
Calculates the hydrogen production efficiency relative to its lower heating value

Parameters data (*simulation results*) –

Returns float: h2 production efficiency

hydrogen_outflow_plotting()

hydrogen_production_plotting()

plot()

power_auxilliaries_1_plotting()

power_auxilliaries_2_plotting()

pressures_plotting()

soh(*index=- 1*)

soh_plotting()

temperature_plotting()

title = 'Electrolyzer results'

class FuelCellTechnicalEvaluation(*data, config, path*)

Bases: *simses.analysis.evaluation.technical.technical_evaluation.TechnicalEvaluation*

current_dens_plotting()

current_plotting()

evaluate()

hydrogen_consumption_plotting()

plot()

power_plotting()

pressures_plotting()

temperature_plotting()

title = 'Fuel cell results'

class HydrogenTechnicalEvaluation(*data, config, path*)

Bases: *simses.analysis.evaluation.technical.technical_evaluation.TechnicalEvaluation*

current_dens_el_plotting()

current_el_plotting()

degradation_plotting()

evaluate()

fulfillment_plotting()

h2_production_efficiency_lhv()

Calculates the hydrogen production efficiency relative to its lower heating value

Parameters **data** (*simulation results*) –

Returns float: h2 production efficiency

hydrogen_outflow_plotting()

hydrogen_production_plotting()

plot()

power_auxilliaries_1_el_plotting()

power_auxilliaries_2_el_plotting()

pressures_el_plotting()

soh(*index=- 1*)

soh_plotting()

temperature_el_plotting()

title = 'Hydrogen results'

class LithiumIonTechnicalEvaluation(*data*, *config*, *battery_config*, *path*)

Bases: *simses.analysis.evaluation.technical.technical_evaluation.TechnicalEvaluation*

evaluate()

plot()

class RedoxFlowTechnicalEvaluation(*data*, *config*, *path*)

Bases: *simses.analysis.evaluation.technical.technical_evaluation.TechnicalEvaluation*

property coulomb_efficiency

evaluate()

overview_plotting()

plot()

soc_comparison_plotting()

temperature_plotting()

title_overview = 'Redox flow results'

title_soc = 'Redox flow SOC difference'

title_temperature = 'Redox flow temperature'

class SiteLevelEvaluation(*data*, *energy_management_data*, *config*, *energy_management_config*, *path*)

Bases: *simses.analysis.evaluation.technical.technical_evaluation.TechnicalEvaluation*

energy_events_above_peak(*power_above_peak*)

evaluate()

static get_average_energy_event_above_peak_from(*energy_events*)

static get_average_power_above_peak_from(*power_above_peak*)

static get_max_energy_event_above_peak_from(*energy_events*)

property grid_power

property intraday_energy_bought

property intraday_energy_sold

intraday_power_plotting()

property max_grid_power

static max_power_above_peak(*power_above_peak*)

plot()

property power_above_peak

property self_consumption_rate

property self_sufficiency

site_level_power_plotting()

title_idm = 'Intraday Power'

title_power = 'Site Level Power'

class SystemTechnicalEvaluation(*data, config, path*)

Bases: *simses.analysis.evaluation.technical.technical_evaluation.TechnicalEvaluation*

close()

evaluate()

property max_generation_dc_power_additional

property max_load_dc_power_additional

plot()

total_acdc_efficiency_charge()

total_acdc_efficiency_discharge()

total_dcdc_efficiency_charge()

total_dcdc_efficiency_discharge()

class TechnicalEvaluation(*data, config*)

Bases: *simses.analysis.evaluation.abstract_evaluation.Evaluation*

TechnicalEvaluation is a special evaluation class for calculating technical KPIs, e.g. efficiency.

property average_fulfillment

Calculates the average fulfillment factor of the system/battery. How often can the battery/system charge/discharge the desired amount of power.

Parameters data (*simulation results*) –

Returns average fulfillment factor

Return type float

property capacity_remaining

property changes_of_sign
Calculates the average number of changes of sign per day

Parameters data (*simulation results*) –

Returns average number of changes of sign per day

Return type float

close()

property depth_of_discharges
Calculates the average depth of cycles in discharge direction

Parameters data (*simulation results*) –

Returns average depth of cycles in discharge direction

Return type float

property energy_swapsign
Calculates the average positive (charged) energy between changes of sign

Parameters data (*simulation results*) –

Returns average charged energy between between changes of sign

Return type float

property energy_throughput

property equivalent_full_cycles
Calculates the number of full-equivalent cycles by dividing the amount of charged energy through the initial capacity

Parameters data (*simulation results*) –

Returns number of full-equivalent cycles

Return type float

evaluate()

property max_soc
Calculates the max SOC of the system/battery

Parameters data (*simulation results*) –

Returns average soc

Return type float

property mean_soc
Calculates the mean SOC of the system/battery

Parameters data (*simulation results*) –

Returns average soc

Return type float

property min_soc
Calculates the min SOC of the system/battery

Parameters **data** (*simulation results*) –

Returns average soc

Return type float

plot()

property resting_times
Calculates the average length of resting time of the system/battery

Parameters **data** (*simulation results*) –

Returns average length of resting time in min

Return type float

property round_trip_efficiency
Calculates the round trip efficiency of the system/battery

Parameters **data** (*simulation results*) –

Returns round trip efficiency

Return type float

Module contents

Submodules

class Evaluation(*data*, *config*, *do_evaluation*)
Bases: `abc.ABC`

Within the evaluation class the analysis of each system and storage technology is conducted. It provides results in form of figures and KPIs. The analysis calculations are done with the help of data object provided which accesses the simulation data.

EXT: str = '.csv'

append_figure(*figure*)

append_result(*evaluation_result*)

append_time_series(*name*, *time_series*)

abstract close()

abstract evaluate()

property evaluation_results

extend_figures(*figures*)

extend_results (*evaluation_results*)

get_data ()

get_figures ()

get_file_name ()

get_files_to_transpose ()

property get_name

abstract plot ()

print_results ()

run ()

property should_be_considered

classmethod transpose_files (*files*)

write_to_batch (*path*, *name*, *run*)

write_to_csv (*path*)

class EvaluationMerger (*result_path*, *config*, *version*)

Bases: `object`

EvaluationMerger merges all evaluations results and figures into one HTML file and opens it after finishing.

OUTPUT_NAME: str = 'Results.html'

merge (*evaluations*)

Writes html file from evaluation results and figures.

Parameters evaluations – List of evaluations.

class Description

Bases: `object`

class Economical

Bases: `object`

CASHFLOW: str = 'Cashflow each year'

DISCOUNT_RATE: str = 'Discount Rate'

class DemandCharges

Bases: `object`

COST_WITHOUT_STORAGE: str = 'Demand charges each year without s

COST_WITH_STORAGE: str = 'Demand charges each year with storag

CYCLE: str = 'Demand Charge Billing Cycle'

INTERVAL: str = 'Demand Charge Average Interval'

PRICE: str = 'Demand Charge Price'

class ElectricityConsumption
Bases: object

ELECTRICITY_COST_GRID: str = 'Electricity cost grid'

ELECTRICITY_COST_RENEWABLE: str = 'Electricity cost renewable'

ELECTRICITY_PRICE: str = 'Electricity price'

TOTAL_ELECTRICITY_COST: str = 'Total electricity Cost'

class FCR
Bases: object

POWER_BID_AVERAGE: str = 'Average FCR Power Bid Each Year'

PRICE_AVERAGE: str = 'Average FCR Price Each Year'

REVENUE_YEARLY: str = 'FCR Revenue Each Year'

INTERNAL_RATE_OF_RETURN: str = 'IRR'

INVESTMENT_COSTS: str = 'Investment Costs'

class Intraday
Bases: object

POWER_AVERAGE: str = 'Average Intraday Power Each Year'

PRICE_AVERAGE: str = 'Average Intraday Revenue Price Each Year

REVENUE_YEARLY: str = 'Intraday Revenue Each Year'

LEVELIZED_COST_OF_STORAGE: str = 'Levelized Cost of Storage'

NET_PRESENT_VALUE: str = 'NPV'

class OperationAndMaintenance
Bases: object

ANNUAl_O_AND_M_COST: str = 'Annual Op. and Maint. Cost '

O_AND_M_COST: str = 'Op. and Maint. Cost each year'

PROFITABILITY_INDEX: str = 'Profitability Index'

RETURN_ON_INVEST: str = 'ROI'

class SCI
Bases: object

COST_ELECTRICITY: str = 'Electricity Costs'

COST_WITHOUT_STORAGE: str = 'Electricity cost each year withou

COST_WITH_STORAGE: str = 'Electricity cost each year with stor

PV_FEED_IN_TARIFF: str = 'PV Feed In Tariff'

class Technical
Bases: object

```
ACDC_EFFICIENCY: str = 'AC_DC efficiency total'

ACDC_EFFICIENCY_CHARGE: str = 'AC_DC efficiency charge'

ACDC_EFFICIENCY_DISCHARGE: str = 'AC_DC efficiency discharge'

COULOMB_EFFICIENCY: str = 'Coulomb efficiency'

DCDC_EFFICIENCY: str = 'DC_DC efficiency total'

DCDC_EFFICIENCY_CHARGE: str = 'DC_DC efficiency charge'

DCDC_EFFICIENCY_DISCHARGE: str = 'DC_DC efficiency discharge'

DEPTH_OF_DISCHARGE: str = 'Avg. depth of cycle for discharge'

ENERGY_ABOVE_PEAK_AVG: str = 'Average energy event above peak'

ENERGY_ABOVE_PEAK_MAX: str = 'Max. energy event above peak'

ENERGY_CHANGES_SIGN: str = 'Pos. energy between changes of sign'

ENERGY_H2_COMPRESSION: str = 'Energy for compression of hydrogen'

ENERGY_H2_DRYING: str = 'Energy for Drying of Hydrogen'

ENERGY_H2_LHV: str = 'Energy of Hydrogen relative to LHV'

ENERGY_H2_REACTION: str = 'Energy for chemical reaction'

ENERGY_IDM_BOUGHT: str = 'Energy bought on intraday market'

ENERGY_IDM_SOLD: str = 'Energy sold on intraday market'

ENERGY_THROUGHPUT: str = 'Energy throughput'

ENERGY_WATER_CIRCULATION: str = 'Energy for water circulation'

ENERGY_WATER_HEATING: str = 'Energy for heating of water'

EQUIVALENT_FULL_CYCLES: str = 'Equivalent full cycles'

FULFILLMENT_AVG: str = 'Avg. Fulfillment Factor'

H2_PRODUCTION_EFFICIENCY_LHV: str = 'Hydrogen production efficienc

MAX_GENERATION_DC_POWER_ADDITIONAL: str = 'Max. generation of addi

MAX_GRID_POWER: str = 'Max. grid power'

MAX_LOAD_DC_POWER_ADDITIONAL: str = 'Max. load of additional dc po

MAX_SOC: str = 'SOC max'

MEAN_SOC: str = 'SOC mean'

MIN_SOC: str = 'SOC min'

NUMBER_CHANGES_SIGNS: str = 'Number of changes of signs per day'

NUMBER_ENERGY_EVENTS: str = 'Number of energy events above peak'

PE_EFFICIENCY: str = 'Power electronics efficiency total'
```

POWER_ABOVE_PEAK_AVG: str = 'Average power above peak'

POWER_ABOVE_PEAK_MAX: str = 'Max. grid power above peak'

REMAINING_CAPACITY: str = 'Remaining capacity'

RESTING_TIME_AVG: str = 'Avg. length of resting times'

ROUND_TRIP_EFFICIENCY: str = 'Efficiency round trip'

SELF_CONSUMPTION_RATE: str = 'Self-consumption rate'

SELF_SUFFICIENCY: str = 'Self-sufficiency'

SOH: str = 'Sate of Health'

TOTAL_H2_PRODUCTION_KG: str = 'Total mass of hydrogen'

TOTAL_H2_PRODUCTION_NM: str = 'Total volume hydrogen'

class EvaluationResult(*description*, *unit*, *value*)

Bases: `object`

Provides a structure for evaluation results in order to organize data management for printing to the console, exporting to csv files, etc..

property description

Description of the result

classmethod get_header()

Returns the header of EvaluationResult as a list of strings.

to_console()

Returns EvaluationResult as a string in a format that is suitable for printing to the console.

to_csv()

Returns EvaluationResult as a list of strings.

property unit

Unit of the result

property value

Value of the result

class Unit

Bases: `object`

EURO: str = 'EUR'

EURO_PER_KW: str = 'EUR / kW'

EURO_PER_KWH: str = 'EUR / kWh'

EURO_PER_KW_DAY: str = 'EUR / kW / d'

EURO_PER_MWH: str = 'EUR / MWh'

KG: str = 'kg'

KILOWATT: str = 'kW'

KWH: str = 'kWh'

MINUTES: str = 'min'

NCM: str = 'Nm^3'

NONE: str = ''

PERCENTAGE: str = '%'

WATT: str = 'W'

Module contents

simses.analysis.test package

Submodules

create_economic_analysis_config(*billing_cycle*)

create_general_config()

test_demand_charge_reduction(*billing_cycle, batt_const*)
Performs a unit test by comparing the expected result for a generic time series with the actual result.

create_economic_analysis_config()

create_general_config()

test_energy_cost_reduction(*batt_const*)
Performs a unit test by comparing the expected result for a generic time series with the actual result.

create_economic_analysis_config()

create_general_config()

test_fcr_revenue_stream(*fcr_power_const*)
Performs a unit test by comparing the expected result for a generic time series with the actual result.

create_economic_analysis_config()

create_general_config()

test_intraday_recharge_revenue_stream(*idm_power_const*)
Performs a unit test by comparing the expected result for a generic time series with the actual result.

create_analysis_config()

create_general_config()

setup_system_data_dict()

test_average_fulfillment(*storage_fulfillment*, *result*)

test_capacity_remaining(*soh*, *result*)

test_changes_of_sign(*storage_power*, *result*)

test_depth_of_discharges(*soc*, *result*)

test_energy_swapsign(*storage_power*, *capacity*, *result*)

test_energy_throughput(*storage_power*, *result*)

test_equivalent_full_cycles(*storage_power*, *capacity*, *result*)

test_max_soc(*soc*, *result*)

test_mean_soc(*soc*, *result*)

test_min_soc(*soc*, *result*)

test_resting_times(*storage_power*, *result*)

test_round_trip_efficiency(*storage_power*, *soc*, *result*)

test_total_acdc_efficiency_charge()

test_total_acdc_efficiency_charge_nan()

test_total_acdc_efficiency_discharge()

test_total_charge_efficiency_inf()

test_total_charge_efficiency_zero()

test_total_dcdc_efficiency_charge()

test_total_dcdc_efficiency_discharge()

Module contents

Submodules

class AnalysisFactory(*path*, *config*, *version*)

Bases: `object`

close()

create_evaluation_merger()

create_evaluations()

class StorageAnalysis(*path*, *config*, *batch_dir*, *version*)

Bases: `object`

StorageAnalysis conducts the analysis of the simulated storage systems by SimSES. For each storage technology as well as for each (sub)system key performance indicators (KPI) are generated and time series are plotted. All information is merged into a

HTML file which opens in the standard browser after analysis is finished. The analysis is configured by analysis.ini in the config package. Additionally, KPIs for comparison between multiple simulations are written to file in batch folder located in results path.

close()
Closing all resources in analysis

run()
Executes analysis for all technologies and systems

get_fractional_years(*start_timestamp*, *end_timestamp*)

get_max_for(*data*)

get_mean_for(*data*)

get_min_for(*data*)

get_negative_values_from(*data*)

get_positive_values_from(*data*)

get_sum_for(*data*)

Module contents

Python Module Index

S