API Documentation OpEnCellS

Release v. 0.2

Fabian Schmid

Sep 06, 2021

CONTENTS:

CHAPTER

ONE

OPEN ENERGY CELL SIMULATION

1.1 OpEnCells

OpEnCellS (Open energy cell simulation) is a open-source modeling framework for the simulation of multi energy carrier supply systems. It covers components of the electricity, heat and chemical sector. The tool is developed by Fabian Schmid, written in Python and follows an object-oriented modeling approach. It enables a detailed simulation of various renewable energy technologies with a temporal resolution of one minute or one hour. Electricity sector components involves models for:

- Photovoltaic
- Wind turbines
- Power components
- Inverter
- Batteries

Heat sector components involves models for:

- Heat pump
- Solarthermal
- Heat storage (perfectly mixed and stratified)

Chemical sector components involves models for:

- Fuel cell
- Electrolyzer
- Hydrogen storage

1.2 Introduction

This API documentation describes the technical content of the simulation framework OpEnCells. It contains instructions about how to effectively use the implemented classes and methods. For an example energy system model it is referred to the OpEnCellS full documentation.

CHAPTER

TWO

SERIALIZABLE MODULE

class serializable.**Serializable**(*file_path=None*)

Bases: object

Methods to make simulation serializable with json format, which makes it possible to automatically store and load components parameters in josn file.

Parameters **file_path** (*string*) – File path where to store/load json file.

Note:

- **Example of how to save battery parameters**
  - anaconda prompt: navigate to base folder of simulation
  - open python
  - from serializable import Serializable
  - from component.battery import Battery
  - Serializable.save(Battery(None,None,None,None,"file path to battery.json"), "filepath to store new battery.json")
- Attention json file is very sensible, manipulate it in spyder or suficated editor as atom.

load(*file_path=None*)

Load component parameter form specified json file to make it attributes of component class.

Parameters **file_path** (*string*) – File path where to load json file from.

save(*file_path=None*)

Save component parameter to json file from component class attributes.

Parameters **file_path** (*string*) – File path where to store json file.

CHAPTER THREE

SIMULATABLE MODULE

class simulatable.**Simulatable**(**childs*)

Bases: `object`

Most central methods to make simulation "simulatable", which means that timestep proceeds after the calculation of all component performance of the current timestep. Simulatable class - Parent class for simulation class and all component classes.

Parameters ***childs** (*class*) – All classes which shall be simulated

---

Note:

- **Class needs to be initialized in simulation class with system component as follows:**
  - **e.g. Simulatable.__init__(self,self.env,self.load,self.pv,self.charger,** self.power_junction, self.battery_management, self.battery)

---

balance()

Method, which updates carrier and checks its energy balance.

Parameters **None** (*None*) –

calculate()

Null methods - stands for for calculation methods of component classes.

Parameters **None** (*None*) –

end()

End Method, which terminates Simulatable with end method for all childs and sets time index back to zero.

Parameters **None** (*None*) –

start()

Start Method, which initialize start method for all childs and sets time index to zero.

Parameters **None** (*None*) –

update()

Method, which updates time index and goes to next simulation step for all childs.

Parameters **None** (*None*) –

CHAPTER

FOUR

DATA_LOADER MODULE

class data_loader.**CSV**

Bases: `object`

Relevant methods of CSV loader in order to load csv file of CAMS Radiation Service, MERRA Weather Data or load profile.

Parameters

- **file_name** (*str*) – File path and name of fiel to be loaded.
- **start** (*ind*) – First timestep of csv file to be loaded.
- **end** (*ind*) – Last timestep of csv file to be loaded.

Note:

- Class is parent class of MeteoIrradiation, MeteoWeather and LoadDemand.

---

get_colomn (*i*)

Extracts specific colomn of loaded Pandas Dataframe by read_csv().

Parameters **i** (*int*) – Colomn to be extracted from Pandas Dataframe __data_set.

Returns **__data_set** – Pandas Series with specified colomn in case loaded pandas Dataframe has multiple colomns.

Return type *Pandas.Series*

read_csv (*file_name*, *start*, *end*)

Loads the csv file and stores it in parameter __data_set

Parameters

- **file_name** (*str*) – File path and name of fiel to be loaded.
- **start** (*int*) – First timestep of csv file to be loaded.
- **end** (*int*) – Last timestep of csv file to be loaded.

Returns **__data_set** – Pandas Dataframe with extracted data rows.

Return type *Pandas.Dataframe*

read_pkl (*file_name*)

Loads the pkl file and stores it in parameter __data_set

Parameters **file_name** (*str*) – File path and name of fiel to be loaded.

Returns **__data_set** – Pandas Dataframe with extracted data rows.

Return type *Pandas.Dataframe*

class `data_loader.`**`LoadDemand`**

Bases: `data_loader.CSV`

Relevant method to load load profile from csv file.

Parameters **`None`** (*None*) –

Returns

- **day_profile** (*pandas.series float*) – [Wh] Pandas series of daily load profile specifies load demand per timestep in watthours.
- **week_profile** (*pandas.series float*) – [Wh] Pandas series of weekly load profile specifies load demand per timestep in watthours.
- **year_profile** (*pandas.series float*) – [Wh] Pandas series of yearly load profile specifies load demand per timestep in watthours.
- **heating_profile** (*pandas.series float*) – [Wh] Pandas series of load profile specifies heating load demand per timestep in watthours.
- **hotwater_profile** (*pandas.series float*) – [Wh] Pandas series of load profile specifies hot water load demand per timestep in watthours.

Note:

- Implemented method is usually integrated in load class to directly load load profile data.
- Heat load demand shall be placed in csv file with heating load in 0.column and hot water heat demand in 1.column.
- Electricty load demand is a single column csv file for a timeframe of a day, week or year.

`get_heating_profile``()`

Returns year load profile

`get_hotwater_profile``()`

Returns year load profile

`get_power_profile``()`

Returns day load profile

class `data_loader.`**`MeteoIrradiation`**

Bases: `data_loader.CSV`

Relevant methods to extracte data from CAMS Radiation Service Dataset.

Parameters **`None`** (*None*) –

Returns

- **time** (*pandas.series int*) – Timestamp of loaded irradiation dataset.
- **irradiance_toa** (*pandas.series float*) – [Wh/m2] Irradiation on horizontal plane at the top of atmosphere.
- **ghi_clear_sky** (*pandas.series float*) – [Wh/m2] Clear sky global irradiation on horizontal plane at ground level.
- **bhi_clear_sky** (*pandas.series float*) – [Wh/m2] Clear sky beam irradiation on horizontal plane at ground level.

- **dhi_clear_sky** (*pandas.series float*) – [Wh/m2] Clear sky diffuse irradiation on horizontal plane at ground level.
- **bni_clear_sky** (*pandas.series float*) – [Wh/m2] Clear sky beam irradiation on mobile plane following the sun at normal incidence.
- **ghi** (*pandas.series float*) – [Wh/m2] Global irradiation on horizontal plane at ground level.
- **bhi** (*pandas.series float*) – [Wh/m2] Beam irradiation on horizontal plane at ground level.
- **dhi** (*pandas.series float*) – [Wh/m2] Diffuse irradiation on horizontal plane at ground level.
- **bni** (*pandas.series float*) – [Wh/m2] Beam irradiation on mobile plane following the sun at normal incidence.
- **reliability** (*pandas.series float*) – [1] Proportion of reliable data in the summarization (0-1).

Note:

- Implemented methods are usually integrated in other classes to directly load irradiance data.
- Examples are the classes environment, load, photovoltaic or battery.
- **Cams irradiance datasets can be downloaded at:**
  – http://www.soda-pro.com/web-services/radiation/cams-radiation-service

---

get_bhi()
Returns [Wh/m2] Beam irradiation on horizontal plane at ground level.

get_bhi_clear_sky()
Returns [Wh/m2] Clear sky beam irradiation on horizontal plane at ground level.

get_bni()
Returns [Wh/m2] Beam irradiation on mobile plane following the sun at normal incidence.

get_bni_clear_sky()
Returns [Wh/m2] Clear sky beam irradiation on mobile plane following the sun at normal incidence.

get_dhi()
Returns [Wh/m2] Diffuse irradiation on horizontal plane at ground level.

get_dhi_clear_sky()
Returns [Wh/m2] Clear sky diffuse irradiation on horizontal plane at ground level.

get_ghi()
Returns [Wh/m2] Global irradiation on horizontal plane at ground level.

get_ghi_clear_sky()
Returns [Wh/m2] Clear sky global irradiation on horizontal plane at ground level.

get_irradiance_toa()
Returns [Wh/m2] Irradiation on horizontal plane at the top of atmosphere.

get_reliability()
Returns [1] Proportion of reliable data in the summarization (0-1).

get_time()
Returns Timestamp of loaded irradiation dataset.

class data_loader.**MeteoWeather**
Bases: *data_loader.CSV*

Relevant methods to extracte data from MERRA Dataset.

Parameters **`None`** (*None*) –

Returns

- **date** (*pandas.seriesint*) – Date with format YYYY-MM-DD.
- **time** (*pandas.series int*) – Time of day with format HH-MM.
- **temperature** (*pandas.series float*) – [K] Ambient temperature at 2 m above ground.
- **humidity** (*pandas.series float*) – [%] Relative humidity at 2 m above ground.
- **wind_speed** (*pandas.series float*) – [m/s] Wind speed at 10 m above ground.
- **wind_direction** (*pandas.series float*) – [°] Wind direction at 10 m above ground (0 means from North, 90 from East. . .).
- **rainfall** (*pandas.series float*) – [mm] Rainfall (= rain depth in mm).
- **snowfall** (*pandas.series float*) – [kg/m2] Snowfall.
- **snow_depth** (*pandas.series float*) – [m] Snow depth.

Note:

- Implemented methods are usually integrated in other classes to directly load weather data.
- Examples are the classes environment, load, photovoltaic or battery.
- **MERRA datasets can be downloaded at:**
  - http://www.soda-pro.com/web-services/meteo-data/merra

`get_air_pressure()`
Returns Pressure (hPa);Pressure at ground level

`get_date()`
Returns Date. format YYYY-MM-DD

`get_humidity()`
Returns Relative humidity (%);Relative humidity at 2 m above ground

`get_rainfall()`
Returns Rainfall (kg/m2);Rainfall (= rain depth in mm)

`get_snow_depth()`
Returns Snow depth (m);Snow depth

`get_snowfall()`
Returns Snowfall (kg/m2);Snowfall

`get_temperature()`
Returns Temperature (K);Temperature at 2 m above ground

`get_time()`
Returns Time of day. format HH-MM

`get_wind_direction()`
Returns Wind direction (deg);Wind direction at 10 m above ground (0 means from North, 90 from East. . .)

`get_wind_speed()`
Returns Wind speed (m/s);Wind speed at 10 m above ground

CHAPTER

FIVE

LOAD_ELECTRICITY MODULE

class load_electricity.**Load_Electricity**

Bases: *simulatable.Simulatable*

Relevant methods to define the simulation load profile power.

Parameters **None** (*None*) –

Note:

- Class data_loader is integrated and its method LoadDemand() to integrate csv load.
- This method is called externally before the central method simulate() of the class simulation is called.

calculate()

Extracts power flow of load profile for each timestep in order to make class simulatable..

Parameters **None** (*None*) –

Returns **power** – [W] Load power flow of timestep in watts.

Return type *float*

CHAPTER

SIX

LOAD_HEAT MODULE

class load_heat.**Load_Heat**(*file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods to define the simulation heat load profile.

Parameters **file_path** (*json*) – To load component parameters (optional).

Note:

- Class data_loader is integrated and its method LoadDemand() to integrate csv load.
- This method is called externally before the central method simulate() of the class simulation is called.
- It defines heat load temperature and flow rate dependent on heat power demand.

calculate()

Extracts power flow, flow temperature and volume flow rate of load profile for each timestep in order to make class simulatable.

Parameters **None** (*None*) –

Returns

- **heating_power** (*float*) – [W] Heating load power flow of timestep in watts.
- **heating_volume_flow_rate** (*float*) – [m3/s] Heating volume flow rate with given flow temperature and heating power.
- **hotwater_power** (*float*) – [W] Hot Water load power flow of timestep in watts.
- **hotwater_volume_flow_rate** (*float*) – [m3/s] Hotwater volume flow rate with given flow temperature and hotwater power.

CHAPTER

SEVEN

ENVIRONMENT MODULE

class environment.**Environment**(*timestep*, *file_path=None*)

Bases: `serializable.Serializable`, `simulatable.Simulatable`

Relevant methods for the calculation of the global irradiation and sun position.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **system_orientation** (*floats*) – [°] Tuble of floats definiing the system oriantation with system azimuth and inclination.
- **system_location** (*floats*) – [°] Tuble of floats defining the system location coordinates with longitude and latitude.

Note:

- **System orientation (tuble of floats)**
  - 1. tuble entry system azimuth in degrees [°]. Panel azimuth from north (0°=north, 90°=east, 180°=south, 270°=west).
  - 2. tuble entry system inclination in degrees [°]. Panel tilt from horizontal.
- **System location (tuble of floats)**
  - 1. tuble entry system longitude in degrees [°]. Positive east of prime meridian, negative west of prime meridian.
  - 2. tuble entry system latitude in degrees [°]. Positive north of equator, negative south of equator.

---

start()

Simulatable method. Loads and calculates all relevant environment data, including total, beam, sky, ground irradiation (pvlib), temperature in [K] and windspeed data in [m/s] sun position (pvlib) and angle of inclination (pvlib).

Parameters **None** (*None*) –

Returns

- **sun_position_pvlib** (*floats*) – [°] Tuble of floats, defining the sun position with its elevation and azimuth angle.
- **sun_aoi_pvlib** (*floats*) – [°] Angle of incidence of the solar vector on the module surface.
- **sun_irradiance_pvlib** (*floats*) – [W/m2] Plane on array irradiation (total, beam, sky, ground).

- **power** (*float*) – [Wh] Total plane on array irradiation.

Note:

- All models are based on pvlib libary version 0.7.1.
- **Solar position calculator (pvlib)**
  - pvlib.solarposition.get_solarposition(time, latitude, longitude, altitude=None, pressure=None, method='nrel_numpy', temperature=12, kwargs)
  - https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.solarposition.get_solarposition.html
  - Further details, compare[1],[2] and[3].
- **Angle of incident calculator (pvlib)**
  - The angle of incidence of the solar vector and the module surface normal.
  - pvlib.irradiance.aoi(surface_tilt, surface_azimuth, solar_zenith, solar_azimuth)
  - https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.irradiance.aoi.html
- **Total, beam, sky diffuse and ground reflected in-plane irradiance**
  - Calculated using the specified sky diffuse irradiance model (pvlib).
  - pvlib.irradiance.get_total_irradiance(surface_tilt, surface_azimuth, solar_zenith, solar_azimuth, dni, ghi, dhi, dni_extra=None, airmass=None, albedo=0.25, surface_type=None, model='isotropic', model_perez='allsitescomposite1990', kwargs)
  - https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.irradiance.get_total_irradiance.html
- **Class data_loader**
  - Integrated and its method MeteoIrradiation() and MeteoWeather() to integrate csv weather data.
  - This method is called externally before the central method simulate() of the class simulation is called.

[1] I. Reda and A. Andreas, Solar position algorithm for solar radiation applications. Solar Energy, vol. 76, no. 5, pp. 577-589, 2004.
[2] I. Reda and A. Andreas, Corrigendum to Solar position algorithm for solar radiation applications. Solar Energy, vol. 81, no. 6, p. 838, 2007.
[3] NREL SPA code: http://rredc.nrel.gov/solar/codesandalgorithms/spa/

CHAPTER

EIGHT

PHOTOVOLTAIC MODULE

class photovoltaic.**Photovoltaic**(*timestep*, *peak_power*, *controller_type*, *env*, *file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods for the calculation of photovoltaic performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **peak_power** (*int*) – [Wp] Installed PV peak power.
- **controller_type** (*string*) – Type of charge controller PWM or MPPT.
- **environment** (*class*) – To get access to solar irradiation, ambient temperature and windspeed.
- **file_name** (*json*) – To load component parameters (optional).

Note:

- Photovoltaic with MPPT assumption or fixed voltage (PWM) is possible.
- Corresponding charge controller type mus be considered manually.
- **Models are based on pvlib libary version 0.7.1.**
  - compare https://pvlib-python.readthedocs.io/en/stable/api.html
- **Photovoltaic model parameters based on SAM libary.**
  - compare https://sam.nrel.gov/photovoltaic/pv-sub-page-2.html
  - latest download available at: https://github.com/NREL/SAM/tree/develop/deploy/libraries

---

calculate()

Calculates and extracts all photovoltaic performance parameters from implemented methods.

Parameters **None** (*None*) –

Returns

- **temperature** (*float*) – [K] Photovoltaic cell temperature, equals temperature_cell.
- **power** (*float*) – [W] Photvoltaic overall power of specified installed array specified by parameter peak_power.
- **peak_power_current** (*float*) – [W] Photovoltaic current peak power assuming power degradation by implemented method phovoltaic_aging()

- **state_of_destruction** (*float*) – [-] Phovoltaic state of destruction as fraction of current and nominal peak power.
- **replacement** (*float*) – [s] Time of replacement in case state_of_destruction equals 1.

Note:

- **Method mainly extracts parameters by calling implemented methods:**
  - get_aging()
  - get_state_of_destruction()

end()

Simulatable method, sets time=0 at end of simulation.

get_aging()

Calculates photovoltaic power degradation and current peak power in Watt [W] assuming aonstat power degradation.

Parameters **None** (-) –

Returns **peak_power_current** – [Wp] Photovoltaic current peak power in watt peak.

Return type *float*

get_power_mppt()

Calculates the Photovoltaic Maximum Power.

Parameters **None** (-) –

Returns **power** – [W] DC photovoltaic mpp power in watts.

Return type *float*

Note:

- **Model is based on NREL's PVWatts DC power model[6].**
  - pvlib.pvsystem.pvwatts_dc(g_poa_effective, temp_cell, pdc0, gamma_pdc, temp_ref=25.0).
  - https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.pvsystem.pvwatts_dc.html.

Could be updated according to: https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.pvsystem.pvwatts_dc.html#pvlib.pvsystem.pvwatts_dc

get_power_pwm()

Calculates the photovoltaic power at given voltage through the VI curve determination with the single diode model and gets parameter for single diode model.

Parameters **None** (-) –

Returns

- **photocurrent** (*float*) – [A] Light-generated current in amperes
- **saturation_current** (*float*) – [A] Diode saturation curent in amperes

[6] A. P. Dobos, "PVWatts Version 5 Manual" http://pvwatts.nrel.gov/downloads/pvwattsv5.pdf (2014).

- **resistance_series** (*float*) – [Ohm] Series resistance in ohms
- **resistance_shunt** (*float*) – [Ohm] Shunt resistance in ohms
- **nNsVth** (*float*) – (numeric) The product of the usual diode ideality factor (n, unitless), number of cells in series (Ns), and cell thermal voltage at specified effective irradiance and cell temperature.
- **current** (*np.ndarray/scalar*) – [A] Photovoltaic current in amperes at given voltage level.

Note:

- **To construct VI curve to determine power at given voltage level.**
  - Is based on model by Jain et al.[2].
  - pvlib.pvsystem.i_from_v(resistance_shunt, resistance_series, nNsVth, voltage, saturation_current, photocurrent, method='lambertw')
  - https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.pvsystem.i_from_v.html#pvlib-pvsystem-i-from-v
- **Get values to construct single diode model.**
  - Is based on five parameter model, by De Soto et al. described in[3].
  - Five values for the single diode equation at effective irradiance and cell temperature can be obtained by calling calcparams_desoto.
  - pvlib.pvsystem.calcparams_desoto(effective_irradiance, temp_cell, alpha_sc, a_ref, I_L_ref, I_o_ref, R_sh_ref, R_s, EgRef=1.121, dEgdT=-0.0002677, irrad_ref=1000, temp_ref=25)
  - https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.pvsystem.calcparams_desoto.html
  - results are returend as tubple of above listed returns.
- **To access pvlib module database and get parameters**
  - modules_list = pvlib.pvsystem.retrieve_sam('CECMod') #'SandiaMod'
  - module = modules_list["NuvoSun_FL0912_100"]
- Further references are[4] and[5].

`get_state_of_destruction()`

Calculates the photovoltaic state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters `None` (-) –

Returns

- **state_of_destruction** (*float*) – [1] Photovoltaic State of destruction with SoD=1 representing a broken component.
- **replacement** (*int*) – [s] Time of photovoltaic component replacement in seconds.

[2] A. Jain, A. Kapoor, "Exact analytical solutions of the parameters of real solar cells using Lambert W-function", Solar Energy Materials and Solar Cells, 81 (2004) 269-277.

[3] W. De Soto et al., "Improvement and validation of a model for photovoltaic array performance", Solar Energy, vol 80, pp. 78-88, 2006.

[4] System Advisor Model web page. https://sam.nrel.gov.

[5] A. Dobos, "An Improved Coefficient Calculator for the California Energy Commission 6 Parameter Photovoltaic Module Model", Journal of Solar Energy Engineering, vol 134, 2012.

Note:

- Replacement time is only set in timeseries array in case of a replacement, otherwise entry is 0.
- In case of replacement current_peak_power is reset to nominal power.

`get_temperature()`

Calculates photovoltaic cell temperature with the Sandia PV Array Performance Model integrated in pvlib.

Parameters **`None`** (-) –

Returns **temperature_cell** – [K] Photovoltaic cell temperature (ATTENTION: not C as specified in pvlib)

Return type *float*

Note:

- The Sandia PV Array Performance Model is based on[1].
- **pvlib.temperature.sapm_cell is called with**
  - **pvlib.temperature.sapm_cell(poa_global, temp_air, wind_speed, a,** b, deltaT, irrad_ref=1000)
  - compare, https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.temperature.sapm_cell.html
- **For numerical values of different module configurations, call**
  - pvlib.temperature.TEMPERATURE_MODEL_PARAMETERS

`start()`

Simulatable method, sets time=0 at start of simulation. Calulates all photovoltaic performance parameters by calling all methods based on pvlib.

Parameters **`None`** (*None*) –

Returns

- **temperature_cell** (*float*) – [K] Photovoltaic cell temperature, by calling method photovoltaic.temperature().
- **power_module** (*float*) – [W] Photvoltaic module power of specified module, by calling methods photovoltaic.power_mppt() or photovoltaic_pwm().

Note:

- Photovoltaic power/temperature is calculated using pvlib libary.
- This libary computes parameters not step by step but all in one for env data.
- Method is equal to environment class, where all env data is loaded all in one.
- Other paranmeters are caculated step by step in the method photovoltaic.calculate().

[1] King, D. et al, 2004, "Sandia Photovoltaic Array Performance Model", SAND Report 3535, Sandia National Laboratories, Albuquerque, NM.

CHAPTER

NINE

POWER_COMPONENT MODULE

class power_component.**Power_Component**(*timestep*, *power_nominal*, *links*, *file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods for the calculation of power components performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **power_nominal** (*int*) – [W] Nominal power of power component in watt.
- **input_link** (*class*) – [-] Class of component which supplies input power.
- **file_path** (*json*) – To load components parameters (optional).

Note:

- Model is based on method by Sauer and Schmid[1].
- **Model can be used for all power components with a power dependent efficiency.**
  - e.g. Charge controllers, BMS, power inverters…

---

calculate()

Calculates all power component performance parameters from implemented methods. Decides weather input_power or output_power method is to be called

Parameters **None** (*None*) –

Returns

- **efficiency** (*float*) – [1] Component efficieincy
- **power** (*float*) – [W] Component input/output power in watts.
- **state_of_destruction** (*float*) – [-] Component state of destruction.
- **replacement** (*float*) – [s] time of replacement in case state_of_destruction equals 1.

end()

Simulatable method, sets time=0 at end of simulation.

get_efficiency_input(*input_link_power*)

Calculates power component efficiency, dependent on Power Output eff(P_out).

Parameters **None** (-) –

[1] D. U. Sauer and H. Schmidt, "Praxisgerechte Modellierung und Abschätzung von Wechselrichter-Wirkungsgraden', in 9. Internationales Sonnenforum - Tagungsband I, 1994, pp. 550–557

Returns **efficiency** – [W] Component efficiency.

Return type *float*

Note:

- Calculated power output is NEGATIVE but fuction can only handle Positive value.
- Therefore first abs(), at the end -

get_efficiency_output (*input_link_power*)

Calculates power component efficiency, dependent on Power Input eff(P_in).

Parameters **None** (-) –

Returns **efficiency** – [W] Component efficiency.

Return type *float*

get_power_input (*input_link_power*)

Calculates power component input power, dependent on Power Output P_in(P_out).

Parameters **None** (-) –

Returns **power** – [W] Component input power in watts.

Return type *float*

Note:

- Calculated power output is NEGATIVE but fuction can only handle Positive value.
- Therefore first abs(), at the end -

get_power_output (*input_link_power*)

Calculates power component output power, dependent on Power Input P_out(P_in).

Parameters **None** (-) –

Returns **power** – [W] Component output power in watts.

Return type *float*

get_state_of_destruction ()

Calculates the component state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **None** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Component state of destruction.
- **replacement** (*int*) – [s] Time of component replacement in seconds.

Note:

- replacement_set stays at last replacement timestep for correct sod calculation after first replacement.

start ()

Simulatable method, sets time=0 at start of simulation.

CHAPTER

TEN

INVERTER MODULE

class inverter.**Inverter**(*timestep*, *power_nominal*, *file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods for the calculation of power components performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **power_nominal** (*int*) – [W] Nominal power of power component in watt.
- **links** (*float*) – [W] Power of linked component.
- **file_path** (*json*) – To load components parameters (optional).

Note:

- Model is based on method by Sauer and Schmid[1].
- **Model can be used for all power components with a power dependent efficiency.**
  - e.g. Charge controllers, BMS, power inverters...

---

calculate()

Calls state of destruction calculation of inverter

Parameters **None** (*None*) –

Returns **None**

Return type *None*

end()

Simulatable method, sets time=0 at end of simulation.

get_power_input()

Calculates power component efficiency and input power (DC), dependent on Power Output eff(P_out).

Parameters **None** (-) –

Returns **efficiency** – [W] Component efficiency.

Return type *float*

Note:

[1] D. U. Sauer and H. Schmidt, "Praxisgerechte Modellierung und Abschätzung von Wechselrichter-Wirkungsgraden', in 9. Internationales Sonnenforum - Tagungsband I, 1994, pp. 550–557

- Calculated power output is NEGATIVE but fuction can only handle Positive value.
- Therefore first abs(), at the end -

---

get_power_output()

Calculates power component efficiency and output power (AC), dependent on Power Input eff(P_in).

Parameters **None** (-) –

Returns **efficiency** – [W] Component efficiency.

Return type *float*

get_state_of_destruction()

Calculates the component state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **None** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Component state of destruction.
- **replacement** (*int*) – [s] Time of component replacement in seconds.

Note:

- replacement_set stays at last replacement timestep for correct sod calculation after first replacement.

---

start()

Simulatable method, sets time=0 at start of simulation.

CHAPTER ELEVEN

INVERTER_STATIC MODULE

class inverter_static.**Inverter**(*timestep*, *power_nominal*, *file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods for the calculation of power components performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **power_nominal** (*int*) – [W] Nominal power of power component in watt.
- **links** (*float*) – [W] Power of linked component.
- **file_path** (*json*) – To load components parameters (optional).

Note:

- Model is based on method by Sauer and Schmid[1].
- **Model can be used for all power components with a power dependent efficiency.**
  - e.g. Charge controllers, BMS, power inverters...

---

calculate()

Calls state of destruction calculation of inverter

Parameters **None** (*None*) –

Returns **None**

Return type *None*

end()

Simulatable method, sets time=0 at end of simulation.

get_power_input()

Calculates power component efficiency and input power (DC), dependent on Power Output eff(P_out).

Parameters **None** (-) –

Returns **efficiency** – [W] Component efficiency.

Return type *float*

Note:

[1] D. U. Sauer and H. Schmidt, "Praxisgerechte Modellierung und Abschätzung von Wechselrichter-Wirkungsgraden', in 9. Internationales Sonnenforum - Tagungsband I, 1994, pp. 550–557

- Calculated power output is NEGATIVE but fuction can only handle Positive value.
- Therefore first abs(), at the end -

---

get_power_output()

Calculates power component efficiency and output power (AC), dependent on Power Input eff(P_in).

Parameters **None** (-) –

Returns **efficiency** – [W] Component efficiency.

Return type *float*

get_state_of_destruction()

Calculates the component state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **None** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Component state of destruction.
- **replacement** (*int*) – [s] Time of component replacement in seconds.

Note:

- replacement_set stays at last replacement timestep for correct sod calculation after first replacement.

---

start()

Simulatable method, sets time=0 at start of simulation.

CHAPTER

TWELVE

BATTERY MODULE

class battery.**Battery**(*timestep*, *capacity_nominal_wh*, *env*, *file_path=None*)

Bases: `serializable.Serializable`, `simulatable.Simulatable`

Relevant methods for the calculation of battery performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **capacity_nominal_wh** (*int*) – [Wh] Installed battery capacity in watthours.
- **environment** (*class*) – To get access to solar irradiation, ambient temperature and windspeed.
- **file_path** (*json*) – To load component parameters (optional).

Note:

- Different battery technologies can be modeled with this generic model approach.
- Model parameter need to be loaded and parametrized externally.
- self.power represents the dis/charge power, while self.power_battery the stored dis/charge battery.

calculate()

Simulatable method. Calculation is done inside energy management of electricty carrier.

end()

Simulatable method, sets time=0 at end of simulation.

get_aging_calendar()

Calculates battery calendar aging according to specified float lifetime.

Parameters **None** (-) –

Returns

- **float_life** (*float*) – [a] Battery float lifetime according to battery temperature.
- **float_life_loss** (*float*) – [Wh] Battery absolute capacity loss per timestep due to calendar aging.

Note:

- Model is based on numerical fitting of float lifetime data given in battery datasheet.

- For detailed description of model parametrization, compare[3].

References

get_aging_cycling()

Calculates battery cycling aging according to micro cycle approach.

Parameters **None** (-) –

Returns

- **cycle_life** (*float*) – [a] Battery cycle lifetime according to last micro cycle.
- **cycle_life_loss** (*float*) – [Wh] Battery absolute capacity loss per timestep due to cycling aging.

Note:

- Cycle life dependent on DoD and temperature if specified
- The model is described in detailed in[4] and[5].

get_charge_discharge_boundary()

Calculates battery charge/discharge boundaries.

Parameters **None** (-) –

Returns **charge_discharge_boundary** – [1] Battery charge/discharge boundary.

Return type *float*

Note:

- The model describes the power-dependent charge and discharge boundaries.
- For a detailed description of the parametrization approach [2].

get_power()

Calculates the battery efficiency & charging/discharging power in Watt.

Parameters **None** (-) –

Returns

- **power** (*float*) – [W] Battery charge/discharge power extracted from the battery.
- **efficiency** (*float*) – [1] Battery charge/discharge efficiency.

Note:

- The model describes a power-dependent efficiency.
- For a detailed description of the parametrization approach [2].

[3] F.Schmid, F.Behrendt "Optimal Sizing of Solar Home Systems: Charge Controller Technology and Its Influence on System Design" Under development.

[4] Narayan et al. "A simple methodology for estimating battery lifetimes in Solar Home System design", IEEE Africon 2017 Proceedings, 2017.

[5] Narayan et al. "Estimating battery lifetimes in Solar Home System design using a practical modelling methodology, Applied Energy 228, 2018.

get_state_of_charge()

Calculates the battery state of charge.

Parameters **None** (-) –

Returns **state_of_charge** – [1] Battery state of charge.

Return type *float*

Note:

- Model is based on simple energy balance using an off-line book-keeping method.
- Considers charge/discharge terminal power, power losses, self-discharge rate.
- For a detailed description of the model[2].

get_state_of_destruction()

Calculates the battery state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **None** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Battery State of destruction with SoD=1 representing a broken component.
- **replacement** (*int*) – [s] Time of battery component replacement in seconds.

Note:

- Replacement time is only set in timeseries array in case of a replacement, otherwise entry is 0.
- In case of replacement current_peak_power is reset to nominal power.

get_temperature()

Calculates the battery temperature in Kelvin.

Parameters **None** (-) –

Returns **temperature** – [K] Battery temperature in Kelvin.

Return type *float*

Note:

- Thermal model is based on general heat blalance and convetcive heat

transport to the environment. - Compare heat balance by[1].

get_voltage()

Calculates battery voltage dependent on battery power and State of charge.

Parameters **None** (-) –

[2] Schmid et al. "An open-source modeling tool for multi-objective optimization of renewable nano/micro-off-grid power supply system", Energy, 2020

[1] Bernardi, E. Pawlikowski, and J. Newman, 'A General Energy Balance for Battery Systems', J. Electrochem. Soc., vol. 132, no. 1, p. 5, 1985.

Returns **voltage** – [V] Battery voltage level.

Return type *float*

Note:

- The model describes the voltage dependent on charge/discharge power and state of charge.

`start()`

Simulatable method, sets time=0 at start of simulation.

CHAPTER THIRTEEN

ELECTROLYZER MODULE

class electrolyzer.**Electrolyzer**(*timestep*, *power_nominal*, *storage_link*, *file_path=None*)

Bases: `serializable.Serializable`, `simulatable.Simulatable`

Relevant methods for the calculation of electrolyzer performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **power_nominal** (*int*) – [W] Installed electrolyzer nominal power.
- **storage_link** (*class*) – Hydrogen storage class.
- **file_path** (*json*) – To load component parameters (optional).

Note:

- Generic class, can be used for modeling of various electrolyzer technologies.
- Currently it reflects PEM electrolyzer.

---

calculate()

Simulatable method. Calculation is done inside energy management of electricty carrier.

end()

Simulatable method, sets time=0 at end of simulation.

get_power()

Calculates the electrolyzer power and efficiencies.

Parameters **None** (-) –

Returns

- **hydrogen_produced_power** (*float*) – [W] Electrolyzer produced hydrogen.
- **heat_produced** (*float*) – [W] Electrolyzer produced heat.
- **efficiency_el** (*float*) – [1] Electrolyzer electric efficiency.
- **efficiency_th** (*float*) – [1] Electrolyzer thermal efficiency.

Note: Model is based on RLI Smooth model, compare[1].

---

[1] https://github.com/rl-institut/smooth/blob/dev/smooth/components/component_pem_electrolyzer.py

References

get_state_of_destruction()

Calculates the electrolyer state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **None** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Component state of destruction.
- **replacement** (*int*) – [s] Time of component replacement in seconds.

Note:

- replacement_set stays at last replacement timestep for correct sod calculation after first replacement.

start()

Simulatable method, sets time=0 at start of simulation.

CHAPTER

FOURTEEN

FUELCELL MODULE

class fuelcell.**Fuelcell**(*timestep*, *power_nominal*, *storage_link*, *file_path=None*)

Bases: `serializable.Serializable`, `simulatable.Simulatable`

Relevant methods for the calculation of Fuel Cell performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **power_nominal** (*int*) – [W] Installed fuel cell nominal power.
- **storage_link** (*class*) – Hydrogen storage class.
- **file_path** (*json*) – To load component parameters (optional).

Note:

- Generic class, can be used for modeling of various fuel cell technologies.
- Currently it reflects PEM fuel cells.

---

calculate()

Simulatable method. Calculation is done inside energy management of electricty carrier.

end()

Simulatable method, sets time=0 at end of simulation.

get_power()

Calculates the fuel cell power and efficiencies.

Parameters **None** (-) –

Returns

- **power_hydrogen** (*float*) – [W] Fuel cell hydrogen power.
- **heat_produced** (*float*) – [W] Fuel cell produced heat.
- **efficiency_el** (*float*) – [1] Fuel cell electric efficiency.
- **efficiency_th** (*float*) – [1] Fuel cell thermal efficiency.

Note: Model is based on RLI Smooth model, compare[1].

---

[1] https://github.com/rl-institut/smooth/blob/dev/smooth/components/component_pem_electrolyzer.py

References

`get_state_of_destruction()`

Calculates the electrolyer state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **`None`** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Component state of destruction.
- **replacement** (*int*) – [s] Time of component replacement in seconds.

Note:

- replacement_set stays at last replacement timestep for correct sod calculation after first replacement.

`start()`

Simulatable method, sets time=0 at start of simulation.

CHAPTER

FIFTEEN

HYDROGEN_STORAGE MODULE

class hydrogen_storage.**Hydrogen_Storage**(*timestep*, *capacity_wh*, *file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods for the calculation of hydrogen storage performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **capacity_wh** (*int*) – [Wh] Installed hydrogen capacity (chemical energy).
- **file_path** (*json*) – To load component parameters (optional).

Note: None

---

calculate()

Simulatable method. Calculation is done inside energy management of electricty carrier.

end()

Simulatable method, sets time=0 at end of simulation.

get_state_of_charge()

Calculates the hydrogen storage state of charge.

Parameters **None** (-) –

Returns **state_of_charge** – [1] Storage state of charge.

Return type *float*

Note:

- Model is based on simple energy balance using an off-line book-keeping method.
- Considers no hydrogen losses.

---

get_state_of_destruction()

Calculates the component state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **None** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Component state of destruction.
- **replacement** (*int*) – [s] Time of component replacement in seconds.

Note:

- replacement_set stays at last replacement timestep for correct sod calculation after first replacement.

start()

Simulatable method, sets time=0 at start of simulation.

CHAPTER

SIXTEEN

HEAT_PUMP MODULE

class heat_pump.**Heat_Pump**(*timestep*, *peak_power_th*, *env*, *file_path=None*)

Bases: `serializable.Serializable`, `simulatable.Simulatable`

Relevant methods for the calculation of heat pump performance.

Parameters

- **timestep** ('int') – [s] Simulation timestep in seconds.
- **peak_power_th** (*int*) – [Wp] Installed wind turbine peak power.
- **env** –
- **file_path** (*json*) – To load component parameters (optional).

Note:

- **Peak thermal power is defined at the point L+2/W35 and 100% speed.**
  - Parameter is used to scale heat pump with given specification power_th (also at L+2/W35)
- Fitting of thermal, electric power and cop is done with temperatures in °C.
- Ambient temperature is loaded in K and therfore transfered inside this class to °C.

calculate()

Simulatable method. Calculation is done inside energy management of heat carrier.

end()

Simulatable method, sets time=0 at end of simulation.

get_coefficient_of_performance()

Calculates the coefficient of performance of heat pump

Parameters **None** (*None*) –

Returns **cop** – [1] Heat pump coefficient of performance.

Return type *float*

Note:

- Calculation is done according to fiting to manufacturer data.

get_power_electric()

Calculates electric power consumption of heat pump

Parameters **`None`** (*None*) –

Returns **power_el** – [W] Heat pump electric power input.

Return type *float*

Note:

- Calculation is done according to fiting to manufacturer data.

`get_power_thermal()`

Calculates thermal power output of heat pump

Parameters **`None`** (*None*) –

Returns **power_th** – [W] Heat pump thermal power output.

Return type *float*

Note:

- Calculation is done according to fiting to manufacturer data.

`get_state_of_destruction()`

Calculates the component state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **`None`** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Component state of destruction.
- **replacement** (*int*) – [s] Time of component replacement in seconds.

Note:

- replacement_set stays at last replacement timestep for correct sod calculation after first replacement.

`start()`

Simulatable method, sets time=0 at start of simulation.

CHAPTER

SEVENTEEN

SOLARTHERMAL MODULE

class solarthermal.**Solarthermal**(*timestep*, *number_collectors*, *env*, *control_type*, *file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods for the calculation of solarthermal collector performance.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **number_collectors** (*int*) – [1] Number of installed solarthermal collectors.
- **environment** (*class*) – To get access to solar irradiation, ambient temperature and windspeed.
- **control_type** (*string*) – Solar pump control algorithm, either *no_control*, *two_point_control* or *pi_control*.
- **file_path** (*json*) – To load component parameters (optional).

Note:

- **Class can be implemented with heat storage**
  - Advantage to better model solarthermal input temperature with heat storage.
- **Solarthermal input temperature:**
  - Can be static value.
  - Or dynamic value from heat storage class (needs to be defined in simulation.py).
- Differential equations can be used to compute collector output and mean temperature.
- **Three different solar pump algorithms implemented:**
  - *no_control*, static input/mean/output temperature and no solar pump control, storage can always be charged till maximum temperature.
  - *two_point_control* and *pi_control*, dynamic/static input temperature and storage pump control.

---

calculate()

Calculates all Solarthermal performance parameters by calling implemented methods

Parameters **None** (*None*) –

Returns None

Return type -

Note:

- According to specified control type implemented methods are called.

`solarthermal_efficiency_iam()`

Calculates collector efficiency with Incidence Angle Modifier.

Parameters `None` (-) –

Returns **efficiency_iam** – [1] Collector efficiency with Incidence Angle Modifier.

Return type *float*

Note:

- **Angle definitions: All angles must be defined in rad.**
  - Sun_elevation: Sun elevation.
  - Sun_azimuth: Sun azimuth (N=0,O=90,S=180,W=270).
  - System_tilt: Collector tilt angle.
  - System_azimuth: Collector azimuth angle (N=0,O=90,S=180,W=270).
  - Sun_aoi: Sun incidence angle on tilted plane.
  - Theta_long: Longitudinal angle on tilted plane.
  - Theta_trans: Transversal angle on tilted plane.

`solarthermal_no_control()`

Defines a static solarthermal algorithm with constant input, mean and output temperature.

Parameters `None` (*None*) –

Returns

- **temperature_input** (*float*) – [K] Collector input temperature, static.
- **temperature_mean** (*float*) – [K] Collector mean temperature, static.
- **temperature_output** (*float*) – [K] Collector output temperature, static.
- **efficiency_iam** (*float*) – [1] Collector efficiency with Incidence Angle Modifier, by calling method solarthermal_efficiency_iam().
- **power_theo** (*float*) – [W] Collector output power based on simple energy yield calc, by calling method solarthermal_power()
- **power_real** (*float*) – [W] Collector output power equals power_theo, due to static collector temperatures.
- **volume_flow_rate** (*float*) – [m3/s] Collector volume flow dependent on solarthermal power.

Note:

- It assumes that heat storage can always be charged.
- Can be used for simulations with hourly resolution.
- Incidence Angle Modifier collector efficiency is used.

solarthermal_pi_control()
Defines a PI control algorithm of the solar pump.

Parameters **None** (*None*) –

Returns

- **efficiency_iam** (*float*) – [1] Collector efficiency with Incidence Angle Modifier, by calling method solarthermal_efficiency_iam().
- **temperature_mean** (*float*) – [K] Collector mean temperature, by calling method solarthermal_temperature_integrale().
- **temperature_output** (*float*) – [K] Collector output temperature, by calling method solarthermal_temperature_integrale().
- **power_theo** (*float*) – [W] Collector output power based on simple energy yield calc, by calling method solarthermal_power()
- **volume_flow_rate** (*float*) – [m3/s] Collector volume flow, variable in order to achieve target output temperature.
- **power_real** (*float*) – [W] Collector output power based on volume flow rate and temperature, dependent on heat storage status.

Note:

- Represents Matched Flow Control.
- Incidence Angle Modifier collector efficiency is used.

solarthermal_power(*efficiency_used*)
Calculates collector output power with basic energy yield equation.

Parameters **efficiency_used** (*float*) – [1] Assumed collector efficiency, Incidence Angle Modifier or Optical efficiency.

Returns

- **power_theo** (*float*) – [W] Collector theoretical output power.
- **efficiency** (*float*) – [W] Collector efficiency.

Note:

- Static solarthermal collector power model based on simple energy balance.
- Power can be calculated with static optical efficiency or iam efficiency.
- Power represents theoretical power output, bcs. it is not dependent on voluem flow rate, which is controlled by solar pump algorithm.
- power theoretical can therefore overestimate collector output power.

solarthermal_temperature_integrale(*efficiency_used*)
Calculates collector temperature distribution.

Parameters **efficiency_used** (*float*) – [1] Assumed collector efficiency, Incidence Angle Modifier or Optical efficiency.

Returns

- **temperature_mean** (*float*) – [K] Collector mean temperature.
- **temperature_output** (*float*) – [K] Collector output temperatur.

Note:

- Function is based on Integral Solarthermal collector model, which is based on instationary, integral energy balance.
- Mean collector temperature is based on assumption of linear temperatur distribution.
- Calculation can be done for different collector efficiencies.

`solarthermal_two_point_control()`

Defines a simple Two-Point-Control algorithm of the solar pump.

Parameters **`None`** (*None*) –

Returns

- **volume_flow_rate** (*float*) – [m3/s] Collector volume flow, static to defined base rate.
- **efficiency_iam** (*float*) – [1] Collector efficiency with Incidence Angle Modifier, by calling method solarthermal_efficiency_iam().
- **temperature_mean** (*float*) – [K] Collector mean temperature, by calling method solarthermal_temperature_integrale().
- **temperature_output** (*float*) – [K] Collector output temperature, by calling method solarthermal_temperature_integrale().
- **power_theo** (*float*) – [W] Collector output power based on simple energy yield calc, by calling method solarthermal_power()
- **power_real** (*float*) – [W] Collector output power based on volume flow rate and temperature, dependent on heat storage status.

Note:

- Method follows simple hysteresis curve with specified temperature delta.
- Incidence Angle Modifier collector efficiency is used.

CHAPTER

EIGHTEEN

PIPE MODULE

class pipe.**Pipe**(*timestep*, *length_pipe*, *env*, *input_link*, *file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods to calculate heat loss and temperature in solarthermal system pipe.

Parameters

- **timestep** (*int*) – [s] Simulation timestep in seconds.
- **length_pipe** (*int*) – [m] Lenght of pipe.
- **environment** (*class*) – To get access to solar irradiation, ambient temperature and windspeed.
- **input_link** (*class*) – Class of component which supplies input flow. Solarthermal output temperature == pipe input temperature.
- **file_path** (*json*) – To load component parameters (optional).

Note:

- Pipe is connecting solarthermal collector with heat storage.
- Differential equation is used to calculate pipe output temperature.
- Is also used as delay element between collector and storage.

calculate()

Calculates all pipe performance parameters by calling implemented methods.

Parameters **None** (*None*) –

Returns

- **temperature_input** (*float*) – [K] Solarthermal input temperature.
- **temperature_pipe_input** (*float*) – [K] Pipe input temperature (equals Soalrthermal output temperature).
- **temperature_output** (*float*) – [K] Pipe output temperature, by calling pipe_temperature_integrale().

Note:

- temperture_input defines still solarthermal temperature input, in order to guarantee smooth solarthermal real pwoer determination.

pipe_temperature_integrale()

Calculates pipe temperature distribution.

Parameters **None** (*None*) –

Returns **temperature_output** – [K] Pipe output temperature.

Return type *float*

Note:

- Integral energy balance over pipe between solarthermal collector and heat storage.

CHAPTER

NINETEEN

AUX_COMPONENT MODULE

class aux_component.**Aux_Component**(*timestep*, *power_nominal*, *file_path=None*)

Bases: `serializable.Serializable`, `simulatable.Simulatable`

Auxiliary heat component to support fluctuating energy technologies for heat supply.

Serializable

Type `class. In order to load/save component parameters in json format`

Simulatable

Type `class. In order` to `get time index` of `each Simulation step`

Parameter()

----

timestep : `int`

[s] Simulation timestep in seconds.

power_nominal : `int`

[W] Nomial component power.

file_path : `json`

To load component parameters (optional).

Note:

- Auxiliary component can define electric heater, heat pump or boiler.
- No detailed modeling of component but amount of energy supplied and costs of supplied energy.
- No partial load operation.

calculate()

Calculates all component performance parameters by calling implemented methods.

Parameters **None** (*None*) –

Returns

- **power** (*float*) – [W] Component power.
- **volume_flow_rate** (*float*) – [m3/s] Component volume flow dependent on operation mode and nominal power.
- **energy_fuel** (*float*) – [Wh] Component consumed fuel energy per timestep.

Note:

- Charge algorithm consists of simple two point algorithm with static offset temperatures.

CHAPTER

TWENTY

HEAT_STORAGE MODULE

class heat_storage.**Heat_storage**(*storage_volume*, *storage_number*, *timestep*, *env*, *file_path=None*)

Bases: *serializable.Serializable*, *simulatable.Simulatable*

Relevant methods to calculate heat storage temperature.

Parameters

storage_volume [*int*] [m3] : Storage volume.

storage_number [*int*] [-] : Number of storages.

timestep [*int*] [s] Simulation timestep in seconds.

environment [*class*] To get access to solar irradiation, ambient temperature and windspeed.

input_link_1: ***class*** Class of component which supplies input flow. (e.g. Pipe output temperature == storage input temperature.)

load_link [*class*] Class of component which defines heat/hot water load.

file_path [*json*] To load component parameters (optional).

calculate()

Simulatable method. Calculation is done inside energy management of heat carrier.

end()

Simulatable method, sets time=0 at end of simulation.

get_state_of_destruction()

Calculates the component state of destruction (SoD) and time of component replacement according to end of life criteria.

Parameters **None** (-) –

Returns

- **state_of_destruction** (*float*) – [1] Component state of destruction.
- **replacement** (*int*) – [s] Time of component replacement in seconds.

Note:

- replacement_set stays at last replacement timestep for correct sod calculation after first replacement.

get_temperature()

Perfect Mixed Heat Storage.

Parameters **`None`** –

Note:

- Assumption that productive components get low static return tempertaure, which comes from low tempertaure niveau of storage.
- This temperature distribution is not modelled but assumed to be always apparent at the bottom of the storage.
- Therefore productive components power is calculated with its static temperature input and not with current mean storage temperature.

`get_temperature_loss``()`

Perfect Mixed Heat Storage.

Parameters **`None`** –

Note:

- self discharge energy los is considered.

`start``()`

Simulatable method, sets time=0 at start of simulation.

CHAPTER TWENTYONE

POWER_JUNCTION MODULE

class power_junction.**Power_Junction**(*input_link_1*, *input_link_2*, *load*)

Bases: `simulatable.Simulatable`

Relevant methods of the power junction to compute all power input and output flows and the resulting battery power flow.

Parameters

- **input_link_1** (*class*) – [-] Component 1 that provides input power to junction
- **input_link_2** (*class*) – [-] Component 2 that provides input power to junction
- **load** (*class*) – [-] Load class to integrate load power flow

Note:

- Power junction can be enlarged with further input and output power flows.

calculate()

Calculates needed battery power to balance input and output power flows of junction.

Parameters **None** (*None*) –

Returns **power** – [W] Power junction power flow to battery.

Return type *float*

CHAPTER

TWENTYTWO

CARRIER MODULE

class carrier.**Carrier_el**(*input_links_el*, *output_links_el*, *battery_link*, *electrolyzer_link*, *fuelcell_link*, *inverter_link*, *grid_link*, *input_links_th*, *output_links_th*, *heat_pump_link*, *heat_storage_link*, *env*)

Bases: `simulatable.Simulatable`

Relevant methods of the power junction to compute all power input and output flows and the resulting battery power flow.

Parameters

- **input_link_1** (*class*) – [-] Component 1 that provides input power to carrier
- **input_link_2** (*class*) – [-] Component 2 that provides input power to carrier
- **output_link_1** (*class*) – [-] Component 1 that provides output power to carrier
- **output_link_2** (*class*) – [-] Component 2 that provides output power to carrier

Note:

- Carrier can be electricity or heat carrier.
- It's input and output flows are in each timestep balanced and itself has no storage cap.
- Power flow sign is defined as carrier inputs (+) and outputs (-).

calculate()

Energy Management system.

Parameters **None** (*None*) –

Returns **None**

Return type *None*

Note:

- Calculates power difference of input and output power flows of carrier.
- Runs battery, fuel cell and electrolyzer EMS according to implemented algorithm.
- Calculates power difference of input and output power flows of carrier.
- Runs storage, heat pump EMS according to implemented algorithm.

ems_battery (*input_link_power*)
Energy Management System for battery. Calculates all battery performance parameters from implemented methods.

Parameters **None** (*None*) –

Returns

- **temperature** (*float*) – [K] Battery temperature in Kelvin.
- **power** (:class:``float`) – [W] Battery charge/discharge power extracted from the battery.
- **efficiency** (*float*) – [1] Battery charge/discharge efficiency.
- **state_of_charge** (*float*) – [1] Battery state of charge.
- **charge_discharge_boundary** (*float*) – [1] Battery charge/discharge boundary.
- **capacity_current_wh** (*float*) – [Wh] Battery capacity of current timestep.
- **state_of_healt** (*float*) – [1] Battery state of health.
- **state_of_destruction** (*float*) – [1] Battery state of destruction.
- **voltage** (*float*) – [V] Battery voltage level.

Note:

- **Method mainly extracts parameters by calling implemented methods of battery class:**
  - battery_temperature()
  - battery_power()
  - battery_state_of_charge()
  - battery_charge_discharge_boundary()
  - battery_voltage()
  - battery_aging_cycling()
  - battery_aging_calendar()

-ATTENTION:

- self.battery.power stays at theoretical power level
- real power dis/charged to battery is stored in self.battery.power_battery.

---

ems_electrolyzer (*input_link_power*)
Energy Management System for electrolyzer.

ems_fuelcell (*output_link_power*)
Energy Management System for fuel cell.

ems_heat_pump ()
Energy Management System for heat pump. Calculates all heat pump performance parameters from implemented methods.

Parameters **None** (*None*) –

Returns None

Return type *None*

end()
Simulatable method, sets time=0 at end of simulation.

start()
Simulatable method, sets time=0 at start of simulation.

CHAPTER

TWENTYTHREE

CARRIER_EL MODULE

class carrier_el.**Carrier_el**(*input_links*, *output_links*, *battery_link*, *electrolyzer_link*, *fuel-cell_link*, *inverter_link*, *grid_link*, *env*)

Bases: `simulatable.Simulatable`

Relevant methods of the power junction to compute all power input and output flows and the resulting battery power flow.

Parameters

- **input_link_1** (*class*) – [-] Component 1 that provides input power to carrier
- **input_link_2** (*class*) – [-] Component 2 that provides input power to carrier
- **output_link_1** (*class*) – [-] Component 1 that provides output power to carrier
- **output_link_2** (*class*) – [-] Component 2 that provides output power to carrier

Note:

- Carrier can be electricity or heat carrier.
- It's input and output flows are in each timestep balanced and itself has no storage cap.
- Power flow sign is defined as carrier inputs (+) and outputs (-).

calculate()

Energy Management system.

Parameters **None** (*None*) –

Returns **None**

Return type *None*

Note:

- Calculates power difference of input and output power flows of carrier.
- Runs battery, fuel cell and electrolyzer EMS according to implemented algorithm.

ems_battery(*input_link_power*)

Energy Management System for battery. Calculates all battery performance parameters from implemented methods.

Parameters **None** (*None*) –

Returns

- **temperature** (*float*) – [K] Battery temperature in Kelvin.
- **power** (:class:``float`) – [W] Battery charge/discharge power extracted from the battery.
- **efficiency** (*float*) – [1] Battery charge/discharge efficiency.
- **state_of_charge** (*float*) – [1] Battery state of charge.
- **charge_discharge_boundary** (*float*) – [1] Battery charge/discharge boundary.
- **capacity_current_wh** (*float*) – [Wh] Battery capacity of current timestep.
- **state_of_healt** (*float*) – [1] Battery state of health.
- **state_of_destruction** (*float*) – [1] Battery state of destruction.
- **voltage** (*float*) – [V] Battery voltage level.

Note:

- **Method mainly extracts parameters by calling implemented methods of battery class:**
  - battery_temperature()
  - battery_power()
  - battery_state_of_charge()
  - battery_charge_discharge_boundary()
  - battery_voltage()
  - battery_aging_cycling()
  - battery_aging_calendar()

-ATTENTION:

- self.battery.power stays at theoretical power level
- real power dis/charged to battery is stored in self.battery.power_battery.

`ems_electrolyzer`(*input_link_power*)
Energy Management System for electrolyzer.

`ems_fuelcell`(*output_link_power*)
Energy Management System for fuel cell.

`end`()
Simulatable method, sets time=0 at end of simulation.

`start`()
Simulatable method, sets time=0 at start of simulation.

CHAPTER TWENTYFOUR

CARRIER_TH MODULE

class carrier_th.**Carrier_th**(*input_links*, *output_links*, *heat_pump_link*, *heat_storage_link*, *env*)

Bases: *simulatable.Simulatable*

Relevant methods of the power junction to compute all power input and output flows and the resulting battery power flow.

Parameters

- **input_link_1** (*class*) – [-] Component 1 that provides input power to carrier
- **input_link_2** (*class*) – [-] Component 2 that provides input power to carrier
- **output_link_1** (*class*) – [-] Component 1 that provides output power to carrier
- **output_link_2** (*class*) – [-] Component 2 that provides output power to carrier

---

Note:

- Carrier can be electricity or heat carrier.
- It's input and output flows are in each timestep balanced and itself has no storage cap.
- Power flow sign is defined as carrier inputs (+) and outputs (-).

---

calculate()

Energy Management system.

Parameters **None** (*None*) –

Returns None

Return type *None*

Note:

- Calculates power difference of input and output power flows of carrier.
- Runs storage, heat pump EMS according to implemented algorithm.

---

ems_heat_pump()

Energy Management System for heat pump. Calculates all heat pump performance parameters from implemented methods.

Parameters **None** (*None*) –

Returns None

Return type *None*

end()
Simulatable method, sets time=0 at end of simulation.

start()
Simulatable method, sets time=0 at start of simulation.

CHAPTER

TWENTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX