smooth

Release 0.3.0

May 17, 2021

Contents:

1 Getting started **1**
- 1.1 Documentation . . . 2
- 1.2 Installing smooth . . . 2
- 1.3 General concept . . . 2
- 1.4 Structure of the SMOOTH module . . . 2
- 1.5 Components . . . 2
- 1.6 Examples . . . 3
- 1.7 Framework . . . 3
- 1.8 Optimization . . . 3
- 1.9 Got further questions on using SMOOTH? . . . 4
- 1.10 License . . . 4

2 smooth.components package **5**
- 2.1 Building a component . . . 5
- 2.2 Artificial costs . . . 5
- 2.3 Foreign states . . . 6
- 2.4 Financials . . . 6
- 2.5 Component - The mother class of all components . . . 7
- 2.6 Air Source Heat Pump . . . 9
- 2.7 Battery . . . 11
- 2.8 Biogas Converter . . . 14
- 2.9 Biogas Steam Methane Reformer with Pressure Swing Adsorption . . . 15
- 2.10 Compressor (Hydrogen) . . . 18
- 2.11 Electric Heater . . . 20
- 2.12 Electrolyzer (alkaline) . . . 22
- 2.13 Electrolyzer Waste Heat (alkaline) . . . 25
- 2.14 Energy Demand from CSV . . . 30
- 2.15 Energy Source from CSV . . . 31
- 2.16 Fuel cell CHP . . . 32
- 2.17 Gas Engine CHP Biogas . . . 36
- 2.18 Gate . . . 40
- 2.19 H2 Refuel Cooling System . . . 40
- 2.20 H2 CHP . . . 42
- 2.21 PEM Electrolyzer . . . 43
- 2.22 Power Converter . . . 47
- 2.23 Sink . . . 48

2.24 Storage H2 . . . 48
2.25 Stratified Thermal Storage . . . 52
2.26 Supply . . . 56
2.27 Trailer Gate . . . 57
2.28 Trailer Gate Cascade . . . 58
2.29 Trailer H2 Delivery . . . 59
2.30 Trailer H2 Delivery Cascade . . . 61
2.31 Trailer H2 Delivery Single . . . 64
2.32 Variable Grid . . . 66
2.33 External Components . . . 66
2.34 H2 Dispenser . . . 67
2.35 Submodules . . . 68
2.36 Module contents . . . 68

3 smooth.examples package **69**
3.1 Submodules . . . 69
3.2 Example Model . . . 69
3.3 Example Model (costs) . . . 70
3.4 Example Model (dict) . . . 76
3.5 Example Model (emissions) . . . 76
3.6 Example Model (external components) . . . 77
3.7 Example Model (plotting dicts) . . . 78
3.8 Run Optimization Example . . . 78
3.9 Run Smooth Example . . . 79
3.10 Module contents . . . 79

4 smooth.framework package **81**
4.1 Subpackages . . . 81
4.2 Submodules . . . 90
4.3 Run SMOOTH . . . 90
4.4 Simulation Parameters . . . 93
4.5 Module contents . . . 94

5 smooth.optimization package **95**
5.1 Subpackages . . . 95
5.2 Submodules . . . 95
5.3 Run Optimization . . . 95
5.4 Module contents . . . 102

6 Indices and tables **103**

Python Module Index **105**

Index **107**

CHAPTER 1

Getting started

SMOOTH stands for "Simulation Model for Optimized Operation and Topology of Hybrid energy systems". This tool serves to minimise costs and maximise the sustainability of dynamic energy systems. The key features of SMOOTH are:

- The real world energy system is reduced to its relevant components
- Detailed modelling of components including non-linear component behaviour, state-dependent component behaviour and tracking arbitrary states of the components
- Stepwise simulation without using perfect foresight
- Parameter optimization possible in combination with a genetic algorithm

- *Documentation*
- *Installing smooth*
- *General concept*
- *Structure of the SMOOTH module*
- *Components*
- *Examples*
- *Framework*
- *Optimization*
- *Got further questions on using SMOOTH?*
- *License*

1.1 Documentation

Full documentation can be found here

1.2 Installing smooth

In order to use SMOOTH, the smooth package and its requirements need to be installed. There is the option to clone the current repository of SMOOTH to your local machine using:

```
git clone https://github.com/rl-institut/smooth
```

The necessary requirements (found in requirements.txt in repository) are installed into a working Python3 environment by:

```
pip install -r requirements.txt
```

SMOOTH is then installed by:

```
python setup.py install
```

You also need to install the solver for oemof. This can be done according to this documentation page.

1.3 General concept

SMOOTH solves an explicitly defined energy system with several components, such as energy sources, electrolyzers, storages etc. The energy system is parameterized with the help of different input parameters such as investment and operating costs as well as site-related time series with a fixed time resolution. While the components and the algorithm executing the simulation are part of SMOOTH, each component creates a valid oemof model for each time step and the system is solved using oemof-solph. The financial costs/revenues and emissions, where the costs are divided into variable costs, CAPEX and OPEX, are tracked for each component individually. After the simulation, all costs/revenues and emissions are transferred to annuities (kg/a and EUR/a, respectively) based on the component lifetimes, and the total system financial and emissions annuities are recorded. The notable states of the components and the energy and mass flows of the system are also recorded and all results can be saved for later use.

An additional functionality of SMOOTH is the optimization (MOEA) which optimizes the topology and operational management of an energy system with regards to ecological and economic target criteria. Key parameters of components are chosen, such as the maximum power output or capacity, and varied in numerous versions of the energy system until the optimal solution/s is/are reached. The specification of the final system/s is/are finally returned as SMOOTH results.

1.4 Structure of the SMOOTH module

The SMOOTH module consists of four sections: components, examples, framework and optimization.

1.5 Components

The *smooth.components package* contains all of the existing components of an energy system that have already been built in SMOOTH, along with any related functions. Input parameters that are defined by the user in the model defi-

nition, or by default values that are specified within the component, are used to calculate and determine the behaviour of the component for each timestep. Within each component, an oemof component is created using the parameters defined or calculated in the SMOOTH component to be used later in the oemof model. Visit the section for detailed information on each of the components and how to build a new component.

1.6 Examples

In order to get a better, applied understanding of how to define a model, and either run a simulation or an optimization, see the examples directory for examples, and the *smooth.examples package* for corresponding explanations.

1.7 Framework

The *smooth.framework package* consists of the main function that runs the SMOOTH simulation framework (the `run_smooth()` function) as well as other functions that are necessary for updating and evaluating the simulation results (in the *smooth.framework.functions package*). An outline and brief description of the available functions in the framework is presented below:

- *`run_smooth()`*: the main function which enables the simulation in SMOOTH, and must be called by the user.
- *`calculate_external_costs()`*: calculates costs for components in the system which are not part of the optimization but their costs should be taken into consideration. This function can be called in the same file as the run_smooth function.
- *`debug()`*: generates debugging information from the results, and prints, plots and saves them. It is called in the run_smooth function if the user sets the *show_debug_flag* parameter as True in the simulation parameters.
- *`load_results()`*: loads the saved results of either a simulation or optimization. Can be called by the user in a file where the results are evaluated.
- *`plot_results()`*: plots results of a SMOOTH run, which can be called after the simulation/optimization results are obtained.
- *`print_results()`*: prints the financial results of a SMOOTH run, which can be called after the simulation/optimization results are obtained.
- *`save_results()`*: saves the results of either a SMOOTH run or an optimization, which can be called after the results are obtained.
- *`update_annuities()`*: calculates and updates the financial and emission annuities for the components used in the system. It is called in the generic Component class, which is used to define each component.
- `update_fitted_costs()`: calculates the fixed costs and fixed emissions of a component. The user can define the dependencies on certain values using a set of specific fitting methods. This function is also called in the generic Component class, which is used to define each component.

1.8 Optimization

The genetic algorithm used for the optimization in SMOOTH is defined in the *smooth.optimization package*, along with instructions on how to use it.

1.9 Got further questions on using SMOOTH?

Contact . . .

1.10 License

CHAPTER 2

smooth.components package

This section first explains how to create a new component and what are their generic properties. Listed below are components that can already be used in an energy system model (see examples directory for the usage of components in an energy system).

2.1 Building a component

In order to build a component, you must do the following:

1. Create a subclass of the mother Component (or External Component) class.
2. In the `__init__()` function, define all parameters that are specific to your component, and set default values.
3. Consider if the component requires variable artificial costs depending on system behaviour. If it does, the method for setting the appropriate costs has to be defined in the `prepare_simulation()` function of the new component.
4. Define any other functions that are specific to your component.
5. All components built in SMOOTH must be created as oemof components to be used in the oemof model (see oemof-solph's component list to choose the best fitting component). Then add the component to your oemof model using the `add_to_oemof_model()` function, defining all of the necessary parameters.
6. If the states of the component need updating after each time step, specifiy these in the `update_states()` function.

2.2 Artificial costs

The oemof framework always solves the system by minimizing the costs. In order to be able to control the system behaviour in a certain way, artificial costs as a concept is introduced. These costs are defined in the components and are used in the oemof model (and therefore have an effect on the cost minimization). While artificial costs are treated the same way as real costs by the oemof solver, they are being neglected in the financial evaluation at the end of the

simulation. Unwanted system behaviour can be avoided by setting high (more positive) artificial costs, while the solver can be incentivised to choose a desired system behaviour by implementing lower (more negative) artificial costs.

2.3 Foreign states

Some component behaviour is dependant on so called foreign states - namely a state or attribute of another component (e.g. the artificial costs of the electricity grid can be dependant on a storage state of charge in order to fill the storage with grid electricity when the storage is below a certain threshold). While the effect of the foreign states is determined in the component itself, the mechanics on how to define the foreign states is the same for each component. Foreign states are always defined by the attributes:

- *fs_component_name*: string (or list of strings for multiple foreign states) of the foreign component
- *fs_attribute_name*: string (or list of strings for multiple foreign states) of the attribute name of the component

If a fixed value should be used as a foreign state, here *fs_component_name* has to be set to None and *fs_attribute_name* has to be set to the numerical value.

2.4 Financials

The costs and revenues are tracked for each component individually. There are three types of costs that are taken into consideration in the energy system, namely capital expenditures (CAPEX), operational expenditures (OPEX) and variable costs. The CAPEX costs are fixed initial investment costs (EUR), the OPEX costs are the yearly operational and maintenance costs (EUR/a) and the variable costs are those that are dependant on the use of the component in the system, such as the cost of buying/selling electricity from/to the grid (EUR/unit).

The nancial analysis is based on annuities of the system. The CAPEX cost of a component for one year is calculated by taking into consideration both the lifetime of the given component and the interest rate, and the OPEX costs remains the same because they are already as annuities. The variable cost annuities for the components are calculated by converting the summed variable costs over the simulation time to the summed variable costs over a one year period. If the simulation time is a year, the variable cost annuities are simply the summed variable costs for a year. The below equations demonstrate how the CAPEX and variable costs are calculated. For more information on the financial analysis and the possible fitting methods for the costs, refer to the update_annuities and update_fitted_cost modules in the *smooth.framework.functions package*.

$$CAPEX_{annuity} = CAPEX \cdot \frac{I \cdot (1+I)^L}{(1+I)^L - 1}$$

$$VC_{annuity} = \sum VC \cdot \frac{365}{S}$$

- $CAPEX_{annuity}$ = CAPEX annuity [EUR/a]
- $CAPEX$ = total CAPEX [EUR]
- I = interest rate [-]
- L = component life time [a]
- $VC_{annuity}$ = annual variable costs [EUR/a]
- VC = total variable costs [EUR]
- S = number of simulation days [days]

2.5 Component - The mother class of all components

The generic component class is the mother class for all of the components. The parameters and functions defined here are inherited by each of the specific components.

class smooth.components.component.**Component**

Bases: object

Parameters

- **component** (*str*) – component type
- **name** (*str*) – specific name of the component (must be different to other component names in the system)
- **life_time** (*numerical*) – lifetime of the component [a]
- **sim_params** (*object*) – simulation parameters such as the interval time and interest rate
- **results** (*dict*) – dictionary containing the main results for the component
- **states** (*dict*) – dictionary containing the varying states for the component
- **variable_costs** (*numeric*) – variable costs of the component [EUR/*]
- **artificial_costs** (*numeric*) – artificial costs of the component [EUR/*] (Note: these costs are not included in the final financial analysis)
- **dependency_flow_costs** (*tuple*) – flow that the costs are dependent on
- **capex** (*dict*) – capital costs
- **opex** (*dict*) – operational and maintenance costs
- **variable_emissions** (*float*) – variable emissions of the component [kg/*]
- **dependency_flow_emissions** (*tuple*) – flow that the emissions are dependent on
- **op_emissions** (*dict*) – operational emission values
- **fix_emissions** (*dict*) – fixed emission values
- **fs_component_name** (*str*) – foreign state component name
- **fs_attribute_name** (*str*) – foreign state attribute name

add_to_oemof_model(*busses*, *model*)

This function adds the specific component to the oemof energy system model and has to be defined for each component.

Parameters

- **busses** (*dict*) – Dict of the virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Raises **NotImplementedError** – NotImplementedError raised if the function is not overwritten in specific component definition.

check_validity()

This function is called immediately after the component object is created and checks if the component attributes are valid.

Raises **ValueError** – Value error raised if the life time is not defined or is less than or equal to 0

generate_results()
Generates the results after the simulation.

Returns Results for the calculated emissions, financials and annuities

get_costs_and_art_costs()
Initialize the total variable costs and art. costs [EUR/*]

Returns The total variable costs (including artificial costs)

get_foreign_state_value(*components*, *index=None*)
Get a foreign state attribute value with the name fs_attribute_name of the component fs_component_name. If the fs_component_name is None and the fs_attribute_name set to a number, the number is given back instead.

Parameters

- **components** (*object*) – List containing each component object
- **index** (*int, optional*) – Index of the foreign state (should be None if there is only one foreign state) [-]

Returns Foreign state value

prepare_simulation(*components*)
Prepares the simulation. If a component has artificial costs, this prepare_simulation function is overwritten in the specific component.

Parameters **components** (*list*) – List containing each component object

Returns If used as a placeholder, nothing will be returned. Else, refer to specific component that uses the prepare_simulation function for further detail.

set_parameters(*params*)
Sets the parameters that have been defined by the user (in the model definition) in the necessary components, overwriting the default parameter values. Errors are raised if: - the given parameter is not part of the component - the dependency flows have not been defined

Parameters **params** (*dict ToDo: make sure of this, maybe list*) – The set of parameters defined in the specific component class

Raises **ValueError** – Value error is raised if the parameter defined by the user is not part of the component, or dependency flows are not defined

Returns None

update_constraints(*busses*, *model_to_solve*)
Sometimes special constraints are required for the specific components, which can be written here. Else, this function is used as placeholder for components without constraints.

Parameters

- **busses** (*dict*) – Dict of the virtual buses used in the energy system
- **model_to_solve** – ToDo: look this up in oemof

Returns If used as a placeholder, nothing will be returned. Else, refer to specific component that uses the update_constraints function for further detail.

update_flows(*results*, *comp_name=None*)
Updates the flows of a component for each time step.

Parameters

- **results** (*object*) – The oemof results for the given time step

- **comp_name** (*str, optional*) – The name of the component - while components can generate more than one oemof model, they sometimes need to give a custom name, defaults to None

Returns updated flow values for each flow in the 'flows' dict

update_states(*results*)
Updates the states, used as placeholder for components without states. If a component has states, this update_states function is overwritten in the specific component.

Parameters **results** (*object*) – oemof results object for the given time step

Returns if used as a placeholder, nothing will be returned. Else, refer to specific component that uses the update_states function for further detail.

update_var_costs()
Tracks the cost and artificial costs of a component for each time step.

Returns New values for the updated variable and artificial costs stored in results['variable_costs'] and results['art_costs'] respectively

update_var_emissions()
Tracks the emissions of a component for each time step.

Returns A new value for the updated emissions stored in results['variable_emissions']

2.6 Air Source Heat Pump

This module represents an air source heat pump that uses ambient air and electricity for heat generation, based on oemof thermal's component.

2.6.1 Scope

Air source heat pumps as a means of heat generation extract outside air and increase its temperature using a pump that requires electricity as an input. These components have the potential for the efficient utilization of energy production and distribution in a system, particularly in times of high renewable electricity production coupled with a high thermal demand.

2.6.2 Concept

The basis for the air source heat pump component is obtained from the oemof thermal component, in particular using the cmpr_hp_chiller function to pre-calculate the coefficient of performance. For further information on how this function works, visit oemof thermal's readthedocs site [1].

References

[1] oemof thermal (2019). Compression Heat Pumps and Chillers, Read the Docs: https://oemof-thermal.readthedocs.io/en/latest/compression_heat_pumps_and_chillers.html

class smooth.components.component_air_source_heat_pump.**AirSourceHeatPump**(*params*)
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the air source heat pump component

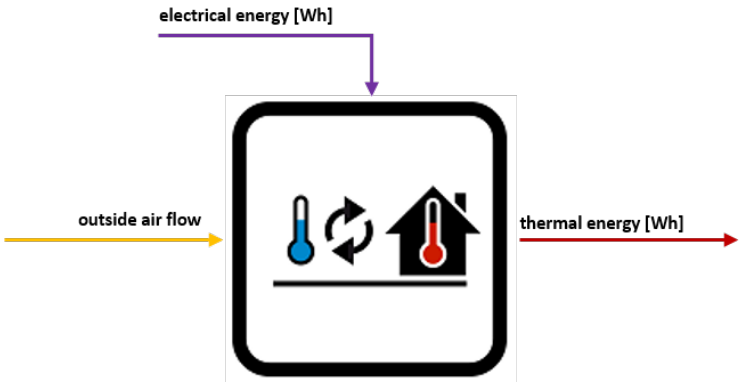

Fig. 1: Fig.1: Simple diagram of an air source heat pump.

- **bus_el** (*str*) – electrical bus input of the heat pump
- **bus_th** – thermal bus output of the heat pump
- **power_max** (*numerical*) – maximum heating output [W]
- **life_time** (*numerical*) – life time of the component
- **csv_filename** (*str*) – csv filename containing the desired timeseries, e.g. 'my_filename.csv'
- **csv_separator** (*str*) – separator of the csv file, e.g. ',' or ';' (default is ',')
- **column_title** (*str or int*) – column title (or index) of the timeseries, default is 0
- **path** (*str*) – path where the timeseries csv file can be located
- **temp_threshold_icing** (*numerical*) – temperature below which icing occurs [K]
- **temp_threshold_icing_C** (*numerical*) – converts to degrees C for oemof thermal function [C]
- **temp_high** (*numerical*) – output temperature from the heat pump [K]
- **temp_high_C** (*numerical*) – converts to degrees C for oemof thermal function [C]
- **temp_high_C_list** (*list*) – converts to list for oemof thermal function
- **temp_low** (*numerical*) – ambient temperature [K]
- **temp_low_C** (*numerical*) – converts to degrees C for oemof thermal function [C]
- **quality_grade** (*numerical*) – quality grade of heat pump [-]
- **mode** (*str*) – can be set to heat_pump or chiller
- **factor_icing** (*numerical*) – COP reduction caused by icing [-]
- **set_parameters** – updates parameter default values (see generic Component class)
- **cops** (*numerical*) – coefficient of performance (pre-calculated by oemof thermal function)

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component from information given in the AirSourceHeatPump class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.7 Battery

This module represents a stationary battery.

2.7.1 Scope

Batteries are crucial in effectively integrating high shares of renewable energy electricity sources in diverse energy systems. They can be particularly useful for off-grid energy systems, or for the management of grid stability and flexibility. This flexibility is provided to the energy system by the battery in times where the electric consumers cannot.

2.7.2 Concept

The battery component has an electricity bus input and output, where factors such as the charging and discharging efficiency, the loss rate, the C-rates and the depth of discharge define the electricity flows.

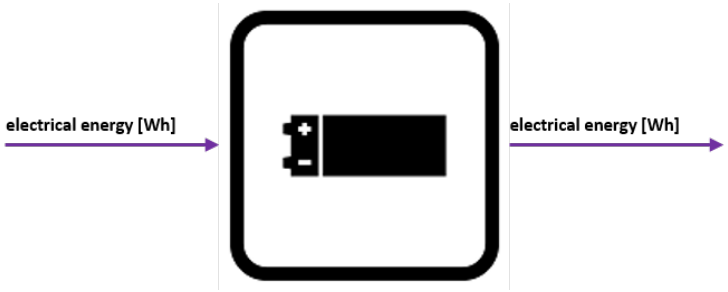

Fig. 2: Fig.1: Simple diagram of a battery storage.

Wanted storage level

Within this component, there is the possibility to choose a wanted storage level that the energy system should try to maintain when it feasibly can. If the state of charge level wanted is defined, the variable artificial costs change depending on whether the storage level is above or below the desired value. If the battery level is too low, the artificial costs of storing electricity into the battery can be reduced and the costs of extracting electricity from the battery can be increased to incentivise the system to maintain the wanted storage level.

Maximum chargeable/dischargeable energy

The maximum power [W] going in to or out of the battery are dependent on the C-rate and the capacity:

$$P_{in,max} = E_{cap} \cdot C_{r,charge}$$
$$P_{out,max} = E_{cap} \cdot C_{r,discharge}$$

- $P_{in,max}$ = maximum power flowing from bus to battery [W]

- E_{cap} = battery capacity [Wh]
- $C_{r,charge}$ = C-Rate for charging [W/Wh]
- $P_{out,max}$ = maximum power flowing from battery to bus [W]
- $C_{r,discharge}$ = C-Rate for discharging [W/Wh]

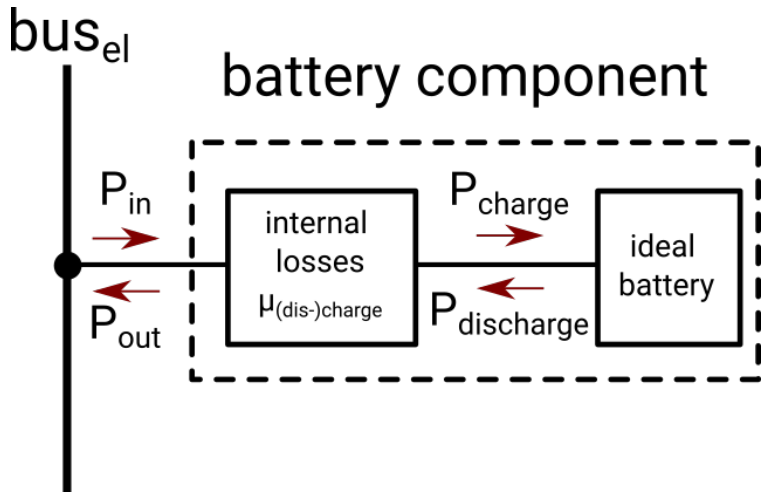

Fig. 3: Fig.2: Diagram of the battery component including losses.

The amount of energy that the battery will be charged or discharged takes the energy losses during the (dis-)charging process into account:

$$P_{charge,max} = P_{in,max} \cdot \mu_{charge}$$
$$P_{out,max} = P_{discharge,max} \cdot \mu_{discharge}$$

- $P_{charge,max}$ = maximum chargeable power at the battery [W]
- μ_{charge} = charging efficiency [-]
- $P_{discharge,max}$ = maximum dischargeable power at the battery [W]
- $\mu_{discharge}$ = discharging efficiency [-]

class smooth.components.component_battery.**Battery**(*params*)
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the battery component
- **bus_in_and_out** (*str*) – electricity bus the battery is connected to
- **battery_capacity** (*numerical*) – battery capacity (assuming all the capacity can be used) [Wh]
- **soc_init** (*numerical*) – initial state of charge [-]
- **efficiency_charge** (*numerical*) – charging efficiency [-]
- **efficiency_discharge** (*numerical*) – discharging efficiency [-]
- **loss_rate** (*numerical*) – loss rate [%/day]
- **symm_c_rate** (*boolean*) – flag to indicate if the c-rate is symmetrical

- **c_rate_symm** (*numerical*) – C-Rate for charging and discharging (only used if symm_c_rate==True) [-/h]
- **c_rate_charge** (*numerical*) – C-Rate for charging [-/h]
- **c_rate_discharge** (*numerical*) – C-Rate for discharging [-/h]
- **soc_min** (*numerical*) – minimal state of charge [-]
- **life_time** (*numerical*) – life time of the component [a]
- **vac_in** (*numerical*) – normal variable artificial costs for charging (in) the battery [EUR/Wh]
- **vac_out** (*numerical*) – normal variable artificial costs for discharging (out) the battery [EUR/Wh]
- **soc_wanted** (*numerical*) – if a soc level is set as wanted, the vac_low costs apply if the capacity is below that level [Wh]
- **vac_low_in** (*numerical*) – variable artificial costs that apply (in) if the capacity level is below the wanted capacity level [EUR/Wh]
- **vac_low_out** (*numerical*) – variable artificial costs that apply (in) if the capacity level is below the wanted capacity level [EUR/Wh]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **soc** (*numerical*) – state of charge [-]
- **p_in_max** (*numerical*) – max. chargeable power [W]
- **p_out_max** (*numerical*) – max. dischargeable power [W]
- **loss_rate** – adjusted loss rate to chosen timestep [%/timestep]
- **current_vac** (*list*) – current artificial costs for input and output [EUR/Wh]

add_to_oemof_model(*busses*, *model*)

Creates an oemof Generic Storage component from the information given in the Battery class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

check_flows()

Checks if there are flows in and out of the battery and if so triggers an AssertionError

Raises ValueError if flows in and out of the battery at the same time are detected

prepare_simulation(*components*)

Prepares the simulation by setting the appropriate artificial costs

Parameters **components** (*list*) – List containing each component object (unused in this component)

update_flows(*results*, *comp_name=None*)

Update flow values for each flow in the 'flows' dict of results

Parameters

- **results** (*object*) – The oemof results for the given time step
- **comp_name** (*str, optional*) – The name of the component - while components can generate more than one oemof model, they sometimes need to give a custom name, defaults to None

update_states (*results*)

Updates the states of the battery component for each time step

Parameters **results** (*object*) – oemof results for the given time step

Returns updated state values for each state in the 'state' dict

2.8 Biogas Converter

This module represents the conversion of a biogas input in m3 to kg, where the biogas composition is defined.

2.8.1 Scope

The biogas converter component is a virtual component, so would not be found in a real life energy system. Its purpose is to transform a biogas bus with m3 units into a biogas bus with kg units. This might be necessary because the Biogas SMR PSA component, for instance, requires a biogas input in kg.

2.8.2 Concept

The biogas converter component takes in a biogas bus as an input and outputs a different biogas bus. The composition of the biogas is defined, as well as the energy content per m3 of biogas.

Biogas composition

The user can determine the desired composition of biogas by stating the percentage share of methane and carbon dioxide in the gas. The default share is chosen to be 75.7% methane, 24.3% carbon dioxide [1]. The lower heating value (LHV) of methane is 13.9 kWh/kg [2], and the molar masses of methane and carbon dioxide are 0.01604 kg/mol and 0.04401 kg/mol, respectively. The method used to calculate the LHV of biogas is the same as in the Gas Engine CHP Biogas component, and the equation used is as follows:

$$LHV_{Bg} = \frac{CH_{4_{share}} \cdot M_{CH_4}}{CH_{4_{share}} \cdot M_{CH_4} + CO_{2_{share}} \cdot M_{CO_2}} \cdot LHV_{CH_4}$$

- LHV_{Bg} = heating value of biogas [kWh/kg]
- $CH_{4_{share}}$ = proportion of methane in biogas [-]
- M_{CH_4} = molar mass of methane [kg/mol]
- $CO_{2_{share}}$ = proportion of carbon dioxide in biogas [-]
- M_{CO_2} = molar mass of carbon dioxide [kg/mol]
- LHV_{CH_4} = heating value of methane [kWh/kg]

References

[1] Braga, L. B. et.al. (2013). Hydrogen production by biogas steam reforming: A technical, economic and ecological analysis, Renewable and Sustainable Energy Reviews. [2] Linde Gas GmbH (2013). Rechnen Sie mit Wasserstoff. Die Datentabelle.

class smooth.components.component_biogas_converter.**BiogasConverter**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the biogas converter component
- **bg_in** (*str*) – input biogas bus
- **bg_out** (*str*) – output biogas bus
- **ch4_share** (*numerical*) – proportion of methane in biogas [-]
- **co2_share** (*numerical*) – proportion of carbon dioxide in biogas [-]
- **kwh_1m3_bg** (*numerical*) – energy content in 1m3 biogas [kWh/m3]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **mol_mass_ch4** (*numerical*) – molar mass of methane [kg/mol]
- **mol_mass_co2** (*numerical*) – molar mass of carbon dioxide [kg/mol]
- **heating_value_ch4** (*numerical*) – heating value of methane [kWh/kg]
- **heating_value_bg** (*numerical*) – heating value of biogas [kWh/kg]

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component from the information given in the BiogasConverter class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.9 Biogas Steam Methane Reformer with Pressure Swing Adsorption

This module represents a biogas steam methane reformer that produces hydrogen, combined with the pressure swing adsorption process to produce 99.9 % pure hydrogen.

2.9.1 Scope

The primary production of hydrogen is currently from the process of steam methane reforming (SMR), using natural gas as the feed material. Biogas can be used as an alternative feed material, which has a similar composition to natural gas. The utilisation of biogas can be beneficial due to biogas being a renewable resource, and its usage can lead to less methane emissions in the atmosphere. Pressure swing adsorption (PSA) is a process used in combination with SMR to purify the output hydrogen stream to a level of approximately 99.9 % [1].

2.9.2 Concept

The biogas SMR PSA component takes in a biogas bus and electricity bus as inputs, with a hydrogen bus output. An oemof Transformer component is chosen for this component, and is illustrated in Figure 1.

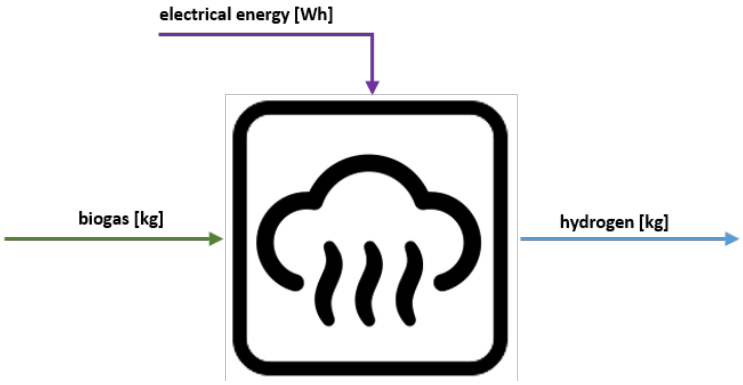

Fig. 4: Fig.1: Simple diagram of a biogas steam methane reformer

Hydrogen production from SMR

The amount of hydrogen produced in SMR from the chosen composition of biogas is calculated based on results from [2]. The default amount of input fuel required to produce 1kg of H2 is 45.977 kWh, and using this value along with the LHV of biogas, the amount of biogas required to produce 1kg of H2 is determined:

$$Bg_{kgH2} = \frac{fuel_{kgH2}}{LHV_{Bg}}$$

- Bg_{kgH2} = biogas required to produce one kg H2 [kg]
- $fuel_{kgH2}$ = specific fuel consumption per kg of H2 produced [kWh/kg]
- LHV_{Bg} = heating value of biogas [kWh/kg]

In order to calculate how much hydrogen will be produced in SMR from the input amount of biogas, the conversion efficiency is calculated:

$$smr_{eff} = \frac{1}{Bg_{kgH2}}$$

- smr_{eff} = conversion efficiency of biogas to hydrogen in SMR process [-]

Hydrogen purification with PSA

The hydrogen produced in SMR contains many impurities such as carbon dioxide and carbon monoxide, and these can be removed using the PSA process. The default efficiency of the PSA process is taken to be 90 % [3], so the overall efficiency of the SMR PSA process is determiend by:

$$overall_{eff} = smr_{eff} \cdot psa_{eff}$$

- $overall_{eff}$ = overall efficiency of biogas to 99.9 % pure hydrogen in SMR and PSA process [-]
- psa_{eff} = efficiency of inpure to pure hydrogen is PSA process [-]

Energy consumption

The default energy consumption of the combined SMR and PSA process per kg of H2 produced is 5.557 kWh/kg [1]. Thus the energy consumption per kg of biogas used is:

$$EC_{kgBg} = \frac{5.557}{Bg_{kgH2}} * 1000$$

- EC_{kgBg} = energy required per kg of biogas used [Wh/kg]

References

[1] Song, C. et.al. (2015). Optimization of steam methane reforming coupled with pressure swing adsorption hydrogen production process by heat integration, Applied Energy. [2] Minh, D. P. et.al. (2018). Hydrogen Production From Biogas Reforming: An Overview of Steam Reforming, Dry Reforming, Dual Reforming and Tri-Reforming of Methane. [3] Air Liquide Engineering & Construction (2021). Druckwechseladsorption Wasserstoffreinigung Rückgewinnung und Reinigung von Wasserstoff durch PSA.

class smooth.components.component_biogas_smr_psa.**BiogasSmrPsa**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the biogas SMR PSA component
- **bus_bg** (*str*) – biogas bus that is the input of the component
- **bus_el** (*str*) – electricity bus that is the input of the component
- **bus_h2** (*str*) – 99.9 % pure hydrogen bus that is the output of the component
- **life_time** (*numerical*) – lifetime of the component [a]
- **input_max** (*numerical*) – maximum biogas input per interval [kg/*]
- **fuel_kwh_1kg_h2** (*numerical*) – specific fuel consumption per kg of H2 produced [kWh/kg]
- **psa_eff** (*numerical*) – efficiency of the PSA process [-]
- **energy_cnsmp_1kg_h2** (*numerical*) – specific energy consumption of the combined SMR and PSA process in terms of hydrogen production [kWh/kg]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **smr_psa_eff** (*numerical*) – total efficiency of biogas to 99.9 % pure hydrogen in SMR and PSA process [-]
- **energy_cnsmp_1kg_bg** (*numerical*) – specific energy consumption of the combined SMR and PSA process in terms of biogas consumption [Wh/kg]

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component using the information given in the BiogasSteamReformer class, to be used in the oemof model

Parameters

- **busses** (*dict*) – buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

prepare_simulation(*components*)
Prepares the simulation by calculating the specific compression energy

Parameters **components** (*list*) – list containing each component object

Returns the specific compression energy [Wh/kg]

2.10 Compressor (Hydrogen)

This module represents a hydrogen compressor.

2.10.1 Scope

A hydrogen compressor is used in energy systems as a means of increasing the pressure of hydrogen to suitable levels for feeding into other components in the system or satisfying energy demands.

2.10.2 Concept

The hydrogen compressor is powered by electricity and intakes a low pressure hydrogen flow while outputting a hgh pressure hydrogen flow. The efficiency of the compressor is assumed to be 88.8%.

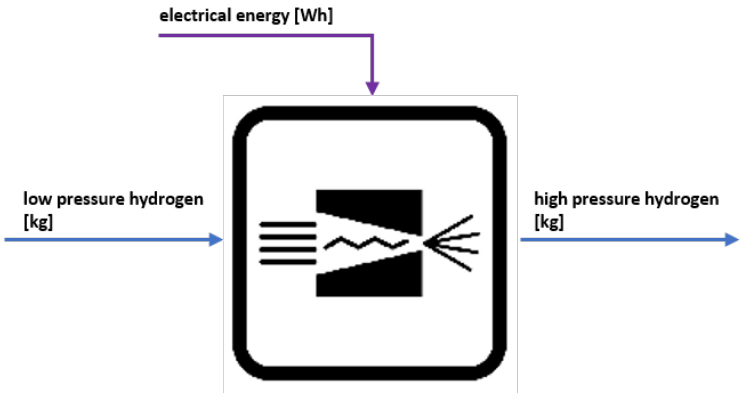

Fig. 5: Fig.1: Simple diagram of a hydrogen compressor.

Specific compression energy

The specific compression energy is calculated by first obtaining the compression ratio:

$$p_{ratio} = \frac{p_{out}}{p_{in}}$$

- p_{ratio} = compression ratio
- p_{out} = outlet pressure [bar]
- p_{in} = inlet pressure [bar]

Then the output temperature is calculated, and the initial assumption for the polytropic exponent is assumed to be 1.6:

$$T_{out} = min(max(T_{in}, T_{in} \cdot p_{ratio}^{\frac{n_{init}-1}{n_{init}}}), T_{in} + 60)$$

- T_{out} = output temperature [K]
- T_{in} = input temperature [K]
- n_{init} = initial polytropic exponent

Then the temperature ratio is calculated:

$$T_{ratio} = \frac{T_{out}}{T_{in}}$$

- T_{ratio} = temperature ratio

Then the polytropic exponent is calculated:

$$n = \frac{1}{1 - \frac{log_{T_{ratio}}}{log_{p,ratio}}}$$

The compressibility factors of the hydrogen entering and leaving the compressor is then calculated using interpolation considering varying temperature, pressure and compressibility factor values (see the calculate_compressibility_factor function). The real gas compressibility factor is calculated using these two values as follows:

$$Z_{real} = \frac{Z_{in} + Z_{out}}{2}$$

- Z_{real} = real gas compressibility factor
- Z_{in} = compressibility factor on entry
- Z_{out} = compressibility factor on exit

Thus the specific compression work is finally calculated:

$$c_{w_1} = \frac{1}{\mu} \cdot R_{H_2} \cdot T_{in} \cdot \frac{n}{n-1} \cdot p_{ratio}^{(\frac{n-1}{n} - 1)} \cdot \frac{Z_{real}}{1000}$$

- c_{w_1} = specific compression work [kJ/kg]
- μ = compression efficiency
- R_{H_2} = hydrogen gas constant

Finally, the specific compression work is converted into the amount of electrical energy required to compress 1 kg of hydrogen:

$$c_{w_2} = \frac{c_{w_1}}{3.6}$$

- c_{w_2} = specific compression energy [Wh/kg]

class smooth.components.component_compressor_h2.**CompressorH2**(*params*)
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the compressor component
- **bus_h2_in** (*str*) – lower pressure hydrogen bus that is an input of the compressor
- **bus_el** (*str*) – electricity bus that is an input of the compressor
- **bus_h2_out** (*str*) – higher pressure hydrogen bus that is the output of the compressor
- **m_flow_max** (*numerical*) – maximum mass flow through the compressor [kg/h]
- **life_time** (*numerical*) – life time of the component [a]

- **temp_in** (*numerical*) – temperature of hydrogen on entry to the compressor [K]
- **efficiency** (*numerical*) – overall efficiency of the compressor [-]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **spec_compression_energy** (*numerical*) – specific compression energy (electrical energy needed per kg H2) [Wh/kg]
- **R** (*numerical*) – gas constant (R) [J/(K*mol)]
- **Mr_H2** (*numerical*) – molar mass of H2 [kg/mol]
- **R_H2** (*numerical*) – specific gas constant for H2 [J/(K*kg)]

add_to_oemof_model(*busses*, *model*)
Creates an oemof Transformer component using the information given in the CompressorH2 class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

prepare_simulation(*components*)
Prepares the simulation by calculating the specific compression energy

Parameters **components** (*list*) – list containing each component object

Returns the specific compression energy [Wh/kg]

update_states(*results*)
Updates the states in the compressor component

Parameters **results** (*object*) – oemof results object for the given time step

Returns updated values for each state in the 'states' dict

smooth.components.component_compressor_h2.**calculate_compressibility_factor**(*p_in*, *p_out*, *temp_in*, *temp_out*)
Calculates the compressibility factor through interpolation.

Parameters

- **p_in** (*numerical*) – inlet pressure [bar]
- **p_out** (*numerical*) – outlet pressure [bar]
- **temp_in** (*numerical*) – inlet temperature of the hydrogen [K]
- **temp_out** (*numerical*) – outlet temperature of the hydrogen [K]

2.11 Electric Heater

A simple electric heater component that converts electricity to heat is created through this module.

2.11.1 Scope

Electric heaters can convert electricity into heat directly with a high efficiency, which can be useful in energy systems with large quantitites of renewable electricity production as well as a heat demand that must be satisfied.

2.11.2 Concept

A simple oemof Transformer component is used to convert the electricity bus into a thermal bus, with a constant efficiency of 98% applied [1].

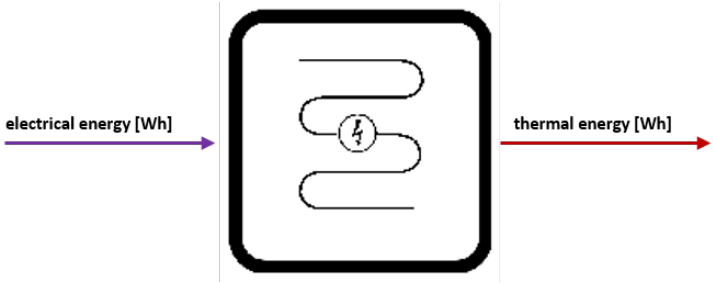

Fig. 6: Fig.1: Simple diagram of an electric heater.

References

[1] Meyers, S. et.al. (2016). Competitive Assessment between Solar Thermal and Photovoltaics for Industrial Process Heat Generation, International Solar Energy Society.

class smooth.components.component_electric_heater.**ElectricHeater**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the electric heater component
- **bus_el** (*str*) – electricity bus that is the input of the electric heater
- **bus_th** (*str*) – thermal bus that is the output of the electric heater
- **power_max** (*numerical*) – maximum thermal output [W]
- **life_time** (*numerical*) – life time of the component [a]
- **efficiency** (*float (0-1)*) – constant efficiency of the heater [-]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component from the information given in the ElectricHeater class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.12 Electrolyzer (alkaline)

This module represents an alkaline electrolyzer that exhibits non-linear behaviour.

2.12.1 Scope

The conversion of electricity into hydrogen can be done through the process of electrolysis. There is a widespread use of alkaline water electrolyzers in dynamic energy systems (involving hydrogen production) due to their simplicity, and providing the electrolyzer with electricity from renewable sources can result in sustainable hydrogen production.

2.12.2 Concept

The alkaline electrolyser intakes an electricity flow and outputs a hydrogen flow. The behaviour of the alkaline electrolyzer is non-linear, which is demonstrated through the use of oemof's Piecewise Linear Transformer component.

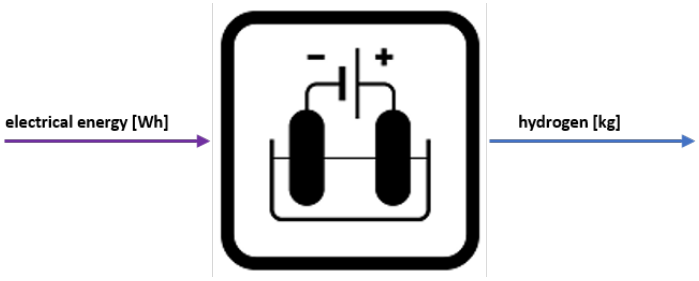

Fig. 7: Fig.1: Simple diagram of an alkaline electrolyzer.

Maximum power

In order to make it possible to define the maximum power of the electrolyser, the number of cells required in the electrolyser is adjusted accordingly. This is achieved by checking how many cells lead to the maximum power at maximum temperature.

Maximum hydrogen production

The maximum amount of hydrogen that can be produced in one time step is determined by the following equation:

$$H_{2,max} = \frac{J_{max} \cdot A_{cell} \cdot t \cdot 60 \cdot z_{cell}}{(2 \cdot F) \cdot \frac{M_{H_2}}{1000}}$$

- $H_{2,max}$ = maximum hydrogen produced in one time step [kg]
- J_{max} = maximum current density [A/cm^2]
- A_{cell} = size of cell surface [cm^2]
- t = interval time
- z_{cell} = number of cells per stack
- F = faraday constant F [As/mol]
- M_{H_2} = molar mass M_H2 [g/mol]

Hydrogen production

Initially, the breakpoints are set up for the electrolyzer conversion of electricity to hydrogen. The breakpoint values for the electric energy are taken in ten evenly spaced incremental steps from 0 to the maximum energy, and the hydrogen production and resulting electrolyzer temperature at each breakpoint is eventually determined.

First, the current density at each breakpoint is calculated (see get_electricity_by_power function). Using this value, the hydrogen mass produced is calculated:

$$H_2 = \frac{I \cdot A_{cell} \cdot t \cdot 60 \cdot z_{cell}}{(2 \cdot F) \cdot \frac{M_{H_2}}{1000}}$$

- H_2 = hydrogen produced in one time step [kg]
- I = current [A]

Varying temperature

The new temperature of the electrolyzer is calculated using Newton's law of cooling. The temperature to which the electrolyzer will heat up to depends on the given current density. Here, linear interpolation is used:

$$T_{aim} = T_{min} + (T_{max} - T_{min}) \cdot \frac{J}{J_{T_{max}}}$$
$$T_{new} = T_{aim} + (T_{old} - T_{aim}) \cdot e^{-t \frac{60}{2310}}$$

- T_{aim} = temperature to which the electrolyser is heating up, depending on current density [K]
- T_{min} = minimum temperature of electrolyzer [K]
- T_{max} = maximum temperature of electrolyzer [K]
- J = current density [A/cm^2]
- $J_{T_{max}}$ = current density at maximum temperature [A/cm^2]
- T_{new} = new temperature of electrolyzer [K]
- T_{new} = new temperature of electrolyzer [K]
- T_{old} = old temperature of electrolyzer [K]
- t = interval time [min]

Additional calculations

For more in depth information on how parameters such as the current density or reversible voltage are calculated, see inside the component for the necessary functions.

class smooth.components.component_electrolyzer.**Electrolyzer**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the electrolyser component
- **bus_el** (*str*) – electricity bus that is the input of the electrolyser
- **bus_h2** (*str*) – hydrogen bus that is the output of the electrolyser
- **power_max** (*numerical*) – maximum power of the electrolyser [W]
- **pressure** (*numerical*) – pressure of hydrogen in the system [Pa]

- **fs_pressure** (*numerical*) – pressure of hydrogen in the system, to be used in other components [bar]
- **temp_init** (*numerical*) – initial electrolyser temperature [K]
- **life_time** (*numerical*) – life time of the component [a]
- **fitting_value_exchange_current_density** (*numerical*) – fitting parameter exchange current density [A/cm^2]
- **fitting_value_electrolyte_thickness** (*numerical*) – thickness of the electrolyte layer [cm]
- **temp_min** (*numerical*) – minimum temperature of the electrolyzer (completely cooled down) [K]
- **temp_max** (*numerical*) – highest temperature the electrolyser can be [K]
- **cur_dens_max** (*numerical*) – maximal current density given by the manufacturer [A/cm^2]
- **cur_dens_max_temp** (*numerical*) – current density at which the maximal temperature is reached [A/cm^2]
- **area_cell** (*numerical*) – size of the cell surface [cm^2]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **interval_time** (*numerical*) – interval time [min]

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component from the information given in the Electrolyzer class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

conversion_fun_ely(*ely_energy*)

Gives out the hydrogen mass values for the electric energy values at the breakpoints

Parameters **ely_energy** (*numerical*) – electric energy values at the breakpoints

Returns The according hydrogen production value [kg]

ely_voltage_u_act(*cur_dens*, *temp*)

Describes the activity losses within the electolyzer

Parameters

- **cur_dens** (*numerical*) – current density [A/cm^2]
- **temp** (*numerical*) – temperature [K]

Returns activation voltage for this node [V]

ely_voltage_u_ohm(*cur_dens*, *temp*)

Takes into account two ohmic losses, one being the # resistance of the electrolyte itself (resistanceElectrolyte) and # other losses like the presence of bubbles (resistanceOther)

Parameters

- **cur_dens** (*numerical*) – current density [A/cm^2]
- **temp** (*numerical*) – temperature [K]

Returns cell voltage loss due to ohmic resistance [V]

ely_voltage_u_rev(*temp*)
Calculates the reversible voltage taking two parts into consideration: the first part takes into account changes of the reversible cell voltage due to temperature changes, the second part due to pressure changes

Parameters **temp** – temperature [K]

Returns reversible voltage [V]

get_cell_temp(*cur_dens*)
Calculates the electrolyzer temperautre for the following time step

Parameters **cur_dens** (*numerical*) – given current density [A/cm^2]

Returns new electrolyzer temperature [K]

get_electricity_by_power(*power*, *this_temp=None*)
Calculates the current density for a given power

Parameters

- **power** (*numerical*) – current power the electrolyzer is operated with [kW]
- **this_temp** (*numerical*) – temperature of the electrolyzer [K]

Returns current density [A/cm^2]

get_mass_and_temp(*energy_used*)
Calculates the mass of hydrogen produced along with the resulting temperature of the electrolyzer for a certain energy

Parameters **energy_used** (*numerical*) – energy value for the next time step [kWh]

Returns produced hydrogen [kg] and the resulting electrolyzer temperature [K]

get_mass_produced_by_current_state(*cur_dens*)
Calculates the hydrogen mass produced by a given current density

Parameters **cur_dens** (*numerical*) – given current density [A/cm^2]

Returns hydrogen mass produced [kg]

update_nonlinear_behaviour()
Updates the nonlinear behaviour of the electrolyser in terms of hydrogen production, as well as the resulting temperature of the electrolyser

update_states(*results*)
Updates the states of the electrolyser component for each time step

Parameters **results** (*object*) – oemof results for the given time step

Returns updated state values for each state in the 'state' dict

2.13 Electrolyzer Waste Heat (alkaline)

This module is created as a subclass of the alkaline Electrolyzer module with the inclusion of a waste heat model.

2.13.1 Scope

The significance of including the heat generation from an electrolyzer in an energy system is that this heat can be utilized for other means (e.g. contributing towards a heat demand) as opposed to wasted. This will be particularly important with the implementation of large scale electrolyzers, where there is the potential to recover large quantities of energy.

2.13.2 Concept

In this component, it is assumed that the alkaline electrolyzer consists of a cylindrical cell stack along with two cylindrical gas separators. It is further assumed that:

- The cell stack height consists of the cells plus two ends. The height of the end of the stack which is not part of the cells has a dependence on the diameter of the cell. The ratio is taken as 7:120 [1]
- The height of an individual cell is in a ratio of 1:75.5 with the cell diameter [2]
- The overall surface area exposed by the gas separators and the pipe communicating them is in a ratio of 1:0.42 with the surface area of the stack [3]

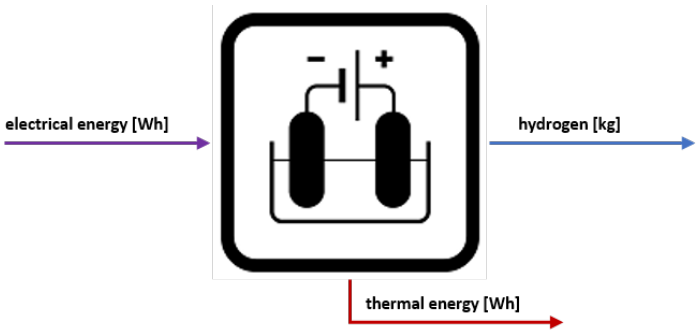

Fig. 8: Fig.1: Simple diagram of an alkaline electrolyzer with waste heat.

The alkaline electrolyzer with waste heat intakes an electrical flow and outputs hydrogen and thermal energy flows. The behaviour of the electrolyzer waste heat model is non-linear, which is demonstrated through the use of oemof's Piecewise Linear Transformer component.

Waste heat

The waste heat equation is derived from the heat balance equation, found in [3]. From this equation, it is shown that the change of the electrolyzer temperature comes from the electrical power input dissipated as heat, without the inclusion of the heat losses to the environment, the heat removed by the cooling water, and the sum of the enthalpy leaving the system with the hydrogen and oxygen streams and the enthalpy gained from the deionized water to warm up the electrolyzer from ambient to operating temperature [3]:

$$C_t \cdot \frac{dT}{dt} = P_{heat} - Q_{loss} - Q_{cooling} - h_j \cdot m_j$$

#ToDo: sort out this equation including derivatives etc.

For the waste heat model in SMOOTH, the heat balance is assumed for stationary conditions. This is because the model only applies when the maximum temperature is reached, and this temperature is then kept constant because of the removal of the waste heat from the system.

The waste heat, which is removed from the electrolyzer by the cooling water, is calculated using the following equation, based on [3]:

$$Q_W = Q_{gen} - Q_L + L + S$$

- Q_W = waste heat
- Q_{gen} = internal heat generation
- Q_L = heat losses to the environment
- L = latent heat
- S = sensible heat

Internal heat generation

The internal heat generation within an electrolyzer is as a result of a greater energy supply to the electrolyzer than is required . This is necessary for reaching high water electrolysis rates [3]. The internal heat gemeration is calculated as follows:

$$Q_{gen} = E_{sup} - H_{2,prod} \cdot HHV_{H_2} \cdot \frac{1e6}{3600}$$

- E_{sup} = total energy supply to the electrolyzer
- $H_{2,prod}$ = the amount of hydrogen produced
- HHV_{H_2} = the higher heating value of hydrogen

Heat losses

In order to calculate the heat losses to the environment, the heat transfer coefficient is first calculated based on [3]. It should be noted that the following equation is specifically to determine the heat transfer coefficient for horizontal cylinders, and since the parts of the alkaline have a cylindrical shape, this equation is used for the alkaline electrolyzer component:

$$h = 1.32 \cdot \frac{\Delta T}{d}^{0.25}$$

The heat losses are then calculated taking into consideration the heat transfer coefficient, the total surface area of the main parts of the electrolyzer (the cell stack and the gas separators) and the temperature difference between the surface of the electrolyzer and the ambient temperature [3]. The equation is as follows:

$$Q_L = A_{sep} \cdot h \cdot (T_{sep} - T_{amb}) + A_{stack} \cdot h \cdot (T_{stack} - T_{amb})$$

- A_{sep} = total surface area of the gas separators
- T_{sep} = separator surface temperature
- T_{amb} = ambient temperature
- A_{stack} = total surface area of the cell stack
- T_{stack} = cell stack surface temperature

Sensible and latent heat

Sensible heat

The sensible heat removed from the system within the H_2 and O_2 streams, as well as the sensible heat required to warm the deionized water from ambient temperature to the stack operating temperature, must be considered when determining the total waste heat. From the known mass of produced hydrogen along with the molar masses of H_2 and O_2, the mass of produced oxygen is determined:

$$m_{O_2} = m_{H_2} \cdot 0.5 \cdot \frac{M_{O_2}}{M_{H_2}}$$

- m_{O_2} = mass of oxygen stream
- m_{H_2} = mass of hydrogen stream
- M_{O_2} = molar mass of oxygen
- M_{H_2} = molar mass of hydrogen

The mass of H_2O is then determined as follows:

$$m_{H_2O} = m_{H_2} + m_{O_2}$$

- m_{H_2O} = mass of water

Thus, the sensible heat is calculated using mass and specific heat:

$$S = \frac{m_{H_2O} \cdot c_{p,H_2O} \cdot -\Delta T - m_{H_2} \cdot c_{p,H_2} \cdot \Delta T + m_{O_2} \cdot c_{p,O_2} \cdot \Delta T}{3.6e6}$$

- c_{p,H_2O} = specific heat of water
- ΔT = the temperature change between the ambient and electrolyzer temperature
- c_{p,H_2} = specific heat of hydrogen
- c_{p,O_2} = specific heat of oxygen

Latent heat

The latent heat is neglected since the mass of H_2O vapor (leaving the system with the oxygen and hydrogen streams) is neglected.

Piecewise Linear Transformer

Currently, the piecewise linear transformer component in oemof can only represent a one-to-one transformation with a singular input and a singular output. Thus, in order to represent the non-linear behaviour of the alkaline electrolyser in the energy system, two oemof components are created for the hydrogen and thermal outputs individually, with a constraint that the electric input flows into each component must always be equal. In this way, the individual oemof components behave as one component.

References

[1] De Silva, Y.S.K. (2017). Design of an Alkaline Electrolysis Stack, University of Agder. [2] Vogt, U.F. et al. (2014). Novel Developments in Alkaline Water Electrolysis, Empa Laboratory of Hydrogen and Energy. [3] Dieguez, P.M. et al. (2008). Thermal Performance of a commercial alkaline water electrolyser: Experimental study and mathematical modeling, Int. J. Hydrogen Energy.

class smooth.components.component_electrolyzer_waste_heat.**ElectrolyzerWasteHeat**(*params*)

Bases: *smooth.components.component_electrolyzer.Electrolyzer*

Parameters

- **param_bus_th** (*dict*) – inclusion of the thermal bus in the parameters dictionary, which was not included in the electrolyzer mother class
- **bus_th** (*str*) – thermal bus that is the output of the electrolyzer
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **interval_time** (*numerical*) – interval time [min]
- **energy_max** (*numerical*) – maximum energy that the electrolyzer can use in one time step [Wh]
- **c_p_H2** (*numerical*) – specific heat of hydrogen at constant pressure [J/(kg*K)]
- **c_p_O2** (*numerical*) – specific heat of oxygen at constant pressure [J/(kg*K)]
- **c_p_H2O** (*numerical*) – specific heat of water at constant pressure [J/(kg*K)]
- **diameter_cell** (*numerical*) – diameter of the electrolyzer cell [m]
- **stack_end_height** (*numerical*) – height of the two stack ends that are not part of the cells, from the perspective of the total stack height [m]
- **height_cell** (*numerical*) – height of an individual cell in relation to the cell diameter [m]
- **height_stack** (*numerical*) – total stack height, which is calculated by taking the cell stack plus the two additional ends of the stack into consideration [m]
- **area_stack** (*numerical*) – external surface area of the electrolyser stack under the assumption that it is cylindrical [m^2]
- **area_separator** (*numerical*) – overall surface area exposed by the gas separators and the pipe communicating them [m^2]
- **model_h2** (*oemof model*) – model created with regards to the hydrogen produced by the electrolyser
- **model_th** (*oemof model*) – model created with regards to the thermal energy produced by the electrolyser

add_to_oemof_model(*busses*, *model*)

Creates two separate oemof Piecewise Linear Transformer components for the hydrogen and thermal production of the electrolyser from information given in the ElectrolyserWasteHeat class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*model*) – oemof model containing the hydrogen production and thermal energy production of the electrolyser

Returns the oemof electrolyzer and electrolyzer thermal components

conversion_fun_ely(*ely_energy*)

Gives out the hydrogen mass values for the electric energy values at the breakpoints

Parameters **ely_energy** (*numerical*) – electric energy values at the breakpoints

Returns The according hydrogen production value [kg]

conversion_fun_thermal (*ely_energy*)
Gives out the thermal energy values for the electric energy values at the breakpoints

Parameters **ely_energy** (*numerical*) – The electric energy values at the breakpoints

Returns The according thermal energy production value [Wh]

get_waste_heat (*energy_used*, *h2_produced*, *new_ely_temp*)
Approximates waste heat production based on calculations of internal heat generation, heat losses to the environment and the sensible and latent heat removed from the system

Parameters

- **energy_used** (*numerical*) – energy consumed by the electrolyser [kWh]
- **h2_produced** (*numerical*) – hydrogen produced by the electrolyser [kg]
- **new_ely_temp** (*numerical*) – resulting temperature of the electrolyser [K]

Returns resulting waste heat produced by the electrolyser [kWh]

sensible_and_latent_heats (*mass_H2*, *new_ely_temp*)
Calculates the sensible and latent heat that has been removed with the hydrogen and oxygen streams leaving the system.

Parameters

- **mass_H2** (*numerical*) – mass of hydrogen [kg]
- **new_ely_temp** (*numerical*) – resulting temperature of the electrolyser [K]

Returns values for the sensible and latent heat

update_constraints (*busses*, *model_to_solve*)
Set a constraint so that the electric inflow of the hydrogen producing and the thermal part are always the same (which is necessary while the piecewise linear transformer cannot have two outputs yet and therefore the two parts need to be separate components).

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model_to_solve** (*model*) – oemof model that will be solved

update_flows (*results*)
Updates the flows of the electrolyser waste heat components for each time step.

Parameters **results** (*object*) – oemof results for the given time step

Returns updated flow values for each flow in the 'flows' dict

update_nonlinear_behaviour ()
Updates the nonlinear behaviour of the electrolyser in terms of hydrogen and thermal energy (waste heat) production, as well as the resulting temperature of the electrolyser

2.14 Energy Demand from CSV

This module represents generic energy demands, which are created through this class by the importation of CSV files.

2.14.1 Scope

The final energy demand component must be satisified by the energy system in the simulations/optimizations.

2.14.2 Concept

The generic energy demand component has one input (the bus is specified by the user), and it requires a demand time series in the form of a CSV file. Optionally, this time series can be created by oemof's demandlib package [1]. This module uses oemof's Sink component.

References

[1] oemof Team (2016). demandlib documentation, https://demandlib.readthedocs.io/en/latest/.

class smooth.components.component_energy_demand_from_csv.**EnergyDemandFromCsv**(*params*)

Bases: *smooth.components.component.Component*

Energy demand components are created through this class by importing csv files.

Parameters

- **name** (*str*) – unique name given to the energy demand component
- **nominal_value** (*numerical*) – value that the timeseries should be multipled by, default is 1
- **csv_filename** – csv filename containing the desired demand timeseries e.g. 'my_demand_filename.csv'
- **csv_separator** (*str*) – separator of the csv file e.g. ',' or ';', default is ','
- **column_title** (*str or int*) – column title (or index) of the timeseries, default is 0
- **path** (*str*) – path where the timeseries csv file can be located
- **bus_in** (*str*) – virtual bus that enters the energy demand component (e.g. the hydrogen bus)
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **data** (*pandas dataframe*) – dataframe containing data from timeseries

add_to_oemof_model(*busses*, *model*)

Creates an oemof Sink component from the information given in the EnergyDemandFromCSV class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.15 Energy Source from CSV

General energy source components are created through this class by importing csv files.

2.15.1 Scope

The energy source components usually represent the various means of renewable energy production in the energy system, which should be efficiently utilised (and sometimes scaled) to avoid excessive energy losses.

2.15.2 Concept

The energy source component is suitable for any energy type once the output bus has been defined as well as a time series in the form of a CSV file, which can be created through oemof's windpowerlib or pvlib, for example [1][2].

References

[1] oemof Team (2016). windpowerlib documentation, https://windpowerlib.readthedocs.io/en/stable/. [2] pvlib Team (2020). pvlib documentation, https://pvlib-python.readthedocs.io/en/v0.7.2/.

class smooth.components.component_energy_source_from_csv.**EnergySourceFromCsv**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the energy source component
- **nominal_value** (*numerical*) – value that the timeseries should be multiplied by, default is 1
- **csv_filename** (*str*) – csv filename containing the desired timeseries, e.g. 'my_filename.csv'
- **csv_separator** (*str*) – separator of the csv file, e.g. ',' or ';' (default is ',')
- **column_title** (*str or int*) – column title (or index) of the timeseries, default is 0
- **path** (*str*) – path where the timeseries csv file can be located
- **bus_out** (*str*) – virtual bus that leaves the energy source component (e.g. the electricity bus)
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **data** (*pandas dataframe*) – dataframe containing data from timeseries

add_to_oemof_model(*busses*, *model*)

Creates an oemof Source component from the information given in the EnergySourceFromCSV class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.16 Fuel cell CHP

This module represents a combined heat and power (CHP) system with a fuel cell, using hydrogen to generate electricity and heat.

2.16.1 Scope

The importance of a fuel cell CHP component in dynamic energy systems is its potential to enable better sector coupling between the electricity and heating sectors, thus less dependence on centralised power systems by offering the ability for localised energy supply [1].

2.16.2 Concept

The fuel cell CHP has a hydrogen bus input along with an electrical bus and thermal bus output. The behaviour of the fuel cell CHP component is non-linear, which is demonstrated through the use of oemof's Piecewise Linear Transformer component.

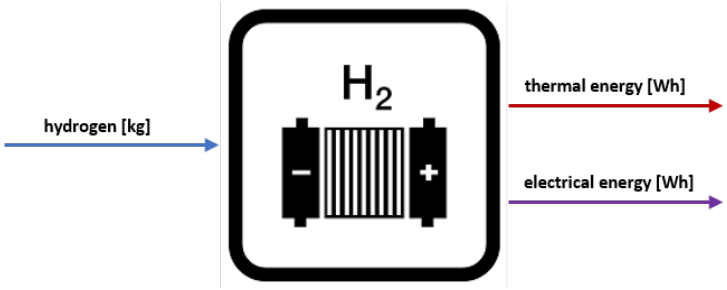

Fig. 9: Fig.1: Simple diagram of a fuel cell CHP.

Efficiency

The efficiency curves for both electrical and thermal energy output according to nominal load which are considered for the fuel cell CHP component are displayed in Figure 2. From the breakpoints, the electrical and thermal production based on the hydrogen consumption and variable efficiency can be obtained. The piecewise linear representation that is actually used in the SMOOTH component is shown in the left image, and the approximated efficiency curve is shown in the right image.

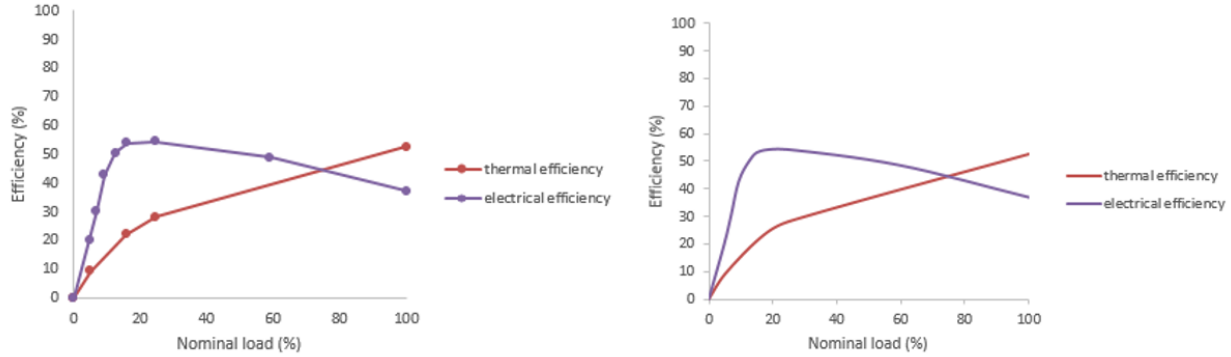

Fig. 10: Fig.2: Piecewise and approximated efficiency of a fuel cell CHP.

Electrical and thermal energy production

In order to calculate the electrical and thermal energy production for each load point, first the maximum hydrogen input is calculated:

$$H_{2,max} = \frac{P_{max}}{LHV_{H_2}} \cdot \mu_{elec_{fullload}}$$

- $H_{2,max}$ = maximum hydrogen input per timestep [kg]
- P_{max} = maximum electrical output power [W]

- $\mu_{elec_{max}}$ = electrical efficiency at full load [-]

Then the load break points for both the electrical and thermal components are converted into how much hydrogen is consumed at each load break point according to the maximum hydrogen input per time step:

$$bp_{H_2,el,i} = bp_{load,el,i} \cdot H_{2,max}$$
$$bp_{H_2,th,i} = bp_{load,th,i} \cdot H_{2,max}$$

- $bp_{H_2,el,i}$ = ith electrical break point in terms of hydrogen consumption [kg]
- $bp_{load,el,i}$ = ith electrical break point in terms of nominal load [-]
- $bp_{H_2,th,i}$ = ith thermal break point in terms of hydrogen consumption [kg]
- $bp_{load,th,i}$ = ith thermal break point in terms of nominal load [-]

From these hydrogen consumption values, the absolute electrical and thermal energy produced at each break point is calculated:

$$E_{el,i} = bp_{H_2,el,i} \cdot \mu_{el,i} \cdot LHV_{H_2} \cdot 1000$$
$$E_{th,i} = bp_{H_2,th,i} \cdot \mu_{th,i} \cdot LHV_{H_2} \cdot 1000$$

- $E_{el,i}$ = ith absolute electrical energy value [Wh]
- $\mu_{el,i}$ = ith electrical efficiency [-]
- $E_{th,i}$ = ith absolute thermal energy value [Wh]
- $\mu_{th,i}$ = ith thermal efficiency [-]

Piecewise Linear Transformer

Currently, the piecewise linear transformer component in oemof can only represent a one-to-one transformation with a singular input and a singular output. Thus, in order to represent the non-linear fuel cell CHP in the energy system, two oemof components are created for the electrical and thermal outputs individually, with a constraint that the hydrogen input flows into each component must always be equal. In this way, the individual oemof components behave as one component.

References

[1] P.E. Dodds et.al. (2015). Hydrogen and fuel cell technologies for heat: A review, International journal of hydrogen energy.

class smooth.components.component_fuel_cell_chp.**FuelCellChp**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the fuel cell CHP component
- **bus_h2** (*str*) – hydrogen bus that is the input of the CHP
- **bus_el** (*str*) – electricity bus that is the output of the CHP
- **bus_th** (*str*) – thermal bus that is the output of the CHP
- **power_max** (*numerical*) – maximum electrical output power [W]
- **life_time** (*numerical*) – lifetime of the component [a]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)

- **heating_value_h2** (*numerical*) – heating value of hydrogen [kWh/kg]
- **bp_load_el** (*list*) – electrical efficiency load break points [-]
- **bp_eff_el** (*list*) – electrical efficiency break points [-]
- **bp_load_th** (*list*) – thermal efficiency load break points [-]
- **bp_eff_th** (*list*) – thermal efficiency break points [-]
- **h2_input_max** (*numerical*) – maximum hydrogen input that leads to maximum electrical energy in Wh [kg]
- **bp_h2_consumed_el** (*list*) – converted electric load points according to maximum hydrogen input per time step [kg]
- **bp_h2_consumed_th** (*list*) – converted thermal load points according to maximum hydrogen input per time step [kg]
- **bp_energy_el** (*list*) – absolute electrical energy values over the load points [Wh]
- **bp_energy_th** (*list*) – absolute thermal energy values over the load points [Wh]
- **bp_h2_consumed_el_half** (*list*) – half the amount of hydrogen that is consumed [kg]
- **bp_h2_consumed_th_half** (*list*) – half the amount of hydrogen that is consumed [kg]
- **model_el** (*model*) – electric model to set constraints later
- **model_th** (*model*) – thermal model to set constraints later

add_to_oemof_model(*busses*, *model*)

Creates two separate oemof Piecewise Linear Transformer components for the electrical and thermal production of the fuel cell CHP from information given in the FuelCellCHP class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*model*) – oemof model containing the electrical energy production and thermal energy production of the fuel cell CHP

Returns tuple of electric and thermal oemof components

get_el_energy_by_h2(*h2_consumption*)

Gets the electrical energy produced by the according hydrogen consumption value.

Parameters **h2_consumption** – hydrogen consumption value [kg]

Returns according electrical energy value [Wh]

get_th_energy_by_h2(*h2_consumption*)

Gets the thermal energy produced by the according hydrogen consumption value.

Parameters **h2_consumption** – hydrogen consumption value [kg]

Returns according thermal energy value [Wh]

update_constraints(*busses*, *model_to_solve*)

Set a constraint so that the hydrogen inflow of the electrical and the thermal part are always the same (which is necessary while the piecewise linear transformer cannot have two outputs yet and therefore the two parts need to be separate components).

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model_to_solve** (*model*) – The oemof model that will be solved

update_flows (*results*)
Updates the flows of the fuel cell CHP components for each time step.

Parameters **results** (*object*) – The oemof results for the given time step

Returns updated flow values for each flow in the 'flows' dict

2.17 Gas Engine CHP Biogas

A combined heat and power (CHP) plant with a gas engine, using biogas to generate electricity and heat is created through this class.

2.17.1 Scope

Biogas CHPs play a significant role in renewable energy systems by using biogas, which has been produced from organic waste material, as a fuel source to produce both electricity and heat. The utilisation of biogas CHPs is beneficial for sector coupling and the minimisation of methane emissions as a result of using up the biogas.

2.17.2 Concept

The biogas CHP component requires a biogas bus input in order to output an electrical and a thermal bus, and oemof's Piecewise Linear Transformer component is chosen to represent the nonlinear efficiencies of the biogas CHP. The method used in this component is similar to the fuel cell CHP component.

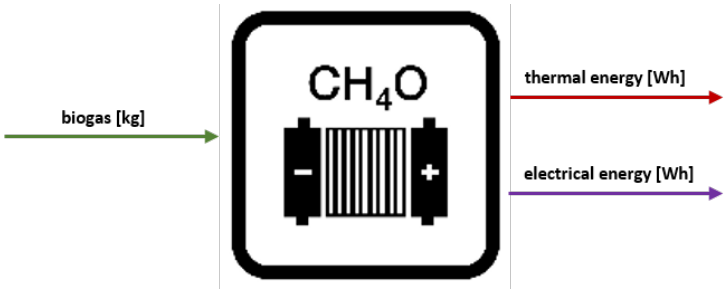

Fig. 11: Fig.1: Simple diagram of a biogas gas engine CHP.

Biogas composition

The user can determine the desired composition of biogas by stating the percentage share of methane and carbon dioxide in the gas (the default is 50-50 % share). The lower heating value (LHV) of methane is 13.9 kWh/kg [1], and the molar masses of methane and carbon dioxide are 0.01604 kg/mol and 0.04401 kg/mol, respectively. The gas composition is given as a mole percentage, and this percentage is transformed into a mass percentage. Finally, the heating value of the biogas is found by multiplying the mass percentage with the LHV of methane, which is demonstrated below:

$$LHV_{Bg} = \frac{CH_{4_{share}} \cdot M_{CH_4}}{CH_{4_{share}} \cdot M_{CH_4} + CO_{2_{share}} \cdot M_{CO_2}} \cdot LHV_{CH_4}$$

- LHV_{Bg} = heating value of biogas [kWh/kg]
- $CH_{4_{share}}$ = proportion of methane in biogas [-]
- M_{CH_4} = molar mass of methane [kg/mol]
- $CO_{2_{share}}$ = proportion of carbon dioxide in biogas [-]
- M_{CO_2} = molar mass of carbon dioxide [kg/mol]
- LHV_{CH_4} = heating value of methane [kWh/kg]

Efficiency

The electrical and thermal production from the CHP is determined by variable efficiencies according to nominal load, and the efficiency curves used in the component can be seen in Figure 2. The piecewise linear representation that is actually used in the SMOOTH component is shown in the left image, and the approximated efficiency curve is shown in the right image.

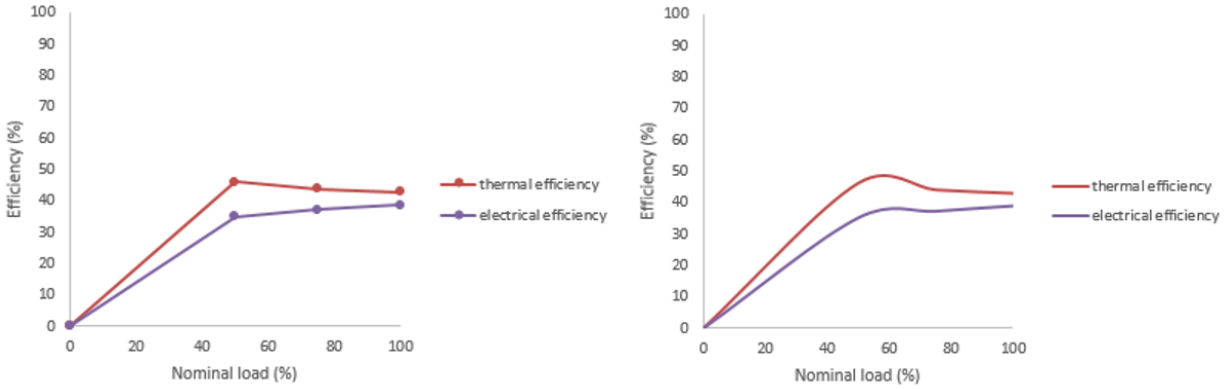

Fig. 12: Fig.2: Piecewise and approximated efficiency of biogas gas engine CHP.

Electrical and thermal energy production

The maximum biogas input is initially calculated so that the electrical and thermal energy production for each load point can be calculated:

$$Bg_{max} = \frac{P_{max}}{LHV_{Bg}} \cdot \mu_{elec_{fullload}}$$

- Bg_{max} = maximum biogas input per timestep [kg]
- P_{max} = maximum electrical output power [W]
- $\mu_{elec_{max}}$ = electrical efficiency at full load [-]

Then the load break points for both the electrical and thermal components are converted into how much biogas is consumed at each load break point according to the maximum biogas input per time step:

$$bp_{Bg,el,i} = bp_{load,el,i} \cdot Bg_{max}$$
$$bp_{Bg,th,i} = bp_{load,th,i} \cdot Bg_{max}$$

- $bp_{Bg,el,i}$ = ith electrical break point in terms of biogas consumption [kg]
- $bp_{load,el,i}$ = ith electrical break point in terms of nominal load [-]

- $bp_{Bg,th,i}$ = ith thermal break point in terms of biogas consumption [kg]
- $bp_{load,th,i}$ = ith thermal break point in terms of nominal load [-]

From these biogas consumption values, the absolute electrical and thermal energy produced at each break point is calculated:

$$E_{el,i} = bp_{Bg,el,i} \cdot \mu_{el,i} \cdot LHV_{Bg} \cdot 1000$$
$$E_{th,i} = bp_{Bg,th,i} \cdot \mu_{th,i} \cdot LHV_{Bg} \cdot 1000$$

- $E_{el,i}$ = ith absolute electrical energy value [Wh]
- $\mu_{el,i}$ = ith electrical efficiency [-]
- $E_{th,i}$ = ith absolute thermal energy value [Wh]
- $\mu_{th,i}$ = ith electrical efficiency [-]

Piecewise Linear Transformer

As stated in the fuel cell CHP component, two seperate oemof components for the electrical and thermal production of the biogas CHP must be created, but they still behave as one component by setting a constraint that the biogas input flows into the two components are always equal.

References

[1] Linde Gas GmbH (2013). Rechnen Sie mit Wasserstoff. Die Datentabelle.

class smooth.components.component_gas_engine_chp_biogas.**GasEngineChpBiogas**(*params*)
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the biogas gas engine CHP component
- **bus_bg** (*str*) – biogas bus input of the CHP
- **bus_el** (*str*) – electrical bus input of the CHP
- **bus_th** (*str*) – thermal bus input of the CHP
- **power_max** (*numerical*) – maximum electrical output power [W]
- **ch4_share** (*numerical*) – proportion of methane in biogas [-]
- **co2_share** (*numerical*) – proportion of carbon dioxide in biogas [-]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **heating_value_ch4** (*numerical*) – heating value of methane [kWh/kg]
- **mol_mass_ch4** (*numerical*) – molar mass of methane [kg/mol]
- **mol_mass_co2** (*numerical*) – molar mass of carbon dioxide [kg/mol]
- **heating_value_bg** (*numerical*) – heating value of biogas [kWh/kg]
- **bp_load_el** (*list*) – electrical efficiency load break points [-]
- **bp_eff_el** (*list*) – electrical efficiency break points [-]
- **bp_load_th** (*list*) – thermal efficiency load break points [-]
- **bp_eff_th** (*list*) – thermal efficiency break points [-]

- **bg_input_max** (*numerical*) – maximum biogas input that leads to the maximum electrical energy in Wh [kg]
- **bp_bg_consumed_el** (*list*) – converted electric load points according to maximum hydrogen input per time step [kg]
- **bp_bg_consumed_th** (*list :param bp_energy_el: absolute electrical energy values over the load points [Wh]*) – converted thermal load points according to maximum hydrogen input per time step [kg]
- **bp_energy_th** (*list*) – absolute thermal energy values over the load points [Wh]
- **bp_bg_consumed_el_half** (*list*) – half the amount of biogas that is consumed [kg]
- **bp_bg_consumed_th_half** (*list*) – half the amount of biogas that is consumed [kg]
- **model_el** (*model*) – electric model to set constraints later
- **model_th** (*model*) – thermal model to set constraints later

add_to_oemof_model (*busses*, *model*)
Creates two separate oemof Piecewise Linear Transformer components for the electrical and thermal production of the biogas CHP from information given in the GasEngineChpBiogas class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*model*) – oemof model containing the electrical energy production and thermal energy production of the biogas CHP

Returns tuple of electric and biogas oemof components

get_electrical_energy_by_bg (*bg_consumption*)
Gets the electrical energy produced by the according biogas production value.

Parameters **bg_consumption** – biogas production value [kg]

Returns according electrical energy value [Wh]

get_thermal_energy_by_bg (*bg_consumption*)
Gets the thermal energy produced by the according biogas production value.

Parameters **bg_consumption** – biogas production value [kg]

Returns according thermal energy value [Wh]

update_constraints (*busses*, *model_to_solve*)
Set a constraint so that the biogas inflow of the electrical and the thermal part are always the same (which is necessary while the piecewise linear transformer cannot have two outputs yet and therefore the two parts need to be separate components).

Parameters

- **busses** –
- **model_to_solve** (*model*) – The oemof model that will be solved

update_flows (*results*)
Updates the flows of the biogas CHP components for each time step.

Parameters **results** (*object*) – The oemof results for the given time step

Returns updated flow values for each flow in the 'flows' dict

2.18 Gate

A gate component is created to transform a specific bus into a more general bus.

2.18.1 Scope

The gate component is a virtual component, so would not be found in a real life energy system, but is used in the framework as a means of transforming a specific bus into a more general bus. As an example, it could be useful in an energy system to initially differentiate between the electricity buses coming out of each renewable energy source, but at some point in the system it could become more useful to have only one generic electricity bus defined.

2.18.2 Concept

An oemof Transformer component is used to convert the chosen input bus into the chosen output bus, with a limitation on the value that can be transformed per timestep by the defined maximum input parameter. Applying an efficiency to the conversion of the input bus to the output bus is optional, with the default value set to 100%.

class smooth.components.component_gate.**Gate**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the gate component
- **max_input** (*numerical*) – maximum value that the gate can intake per timestep
- **bus_in** (*str*) – bus that enters the gate component
- **bus_out** (*str*) – bus that leaves the gate component
- **efficiency** (*numerical*) – efficiency of the gate component
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component from information given in the Gate class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.19 H2 Refuel Cooling System

A component that represents the cooling system in a refuelling station is created through this class.

2.19.1 Scope

As part of the hydrogen refuelling station, a cooling system is required to precool high pressure hydrogen before it is dispensed into the vehicle's tank. This is in order to prevent the tank from overheating.

2.19.2 Concept

An oemof Sink component is used which has one electrical bus input that represents the electricity required to power the cooling system.

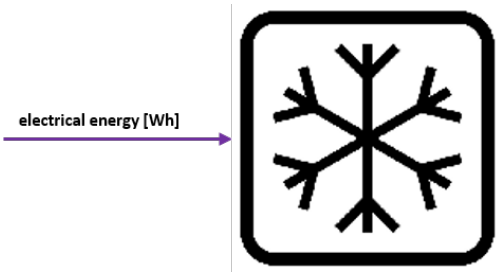

Fig. 13: Fig.1: Simple diagram of a hydrogen refuel cooling system.

The required electricity supply for the cooling system per timestep is calculated by the following equation:

$$E_{elec,i} = \frac{D_{H_2,i} \cdot E_{spec} + E_{standby}}{3.6}$$

- $E_{elec,i}$ = electrical energy required for the ith timestep [Wh]
- $D_{H_2,i}$ = hydrogen demand for the ith timestep [kg]
- E_{spec} = specific energy required relative to the demand [kJ/kg]
- $E_{standby}$ = standby energy required per timestep [kJ/h]

The default specific energy is chosen to be 730 kJ/kg, and the standby energy is chosen to be 8100 kJ/h [find source]. Furthermore, this cooling system component is only necessary if the hydrogen is compressed over 350 bar e.g. to 700 bar for passenger cars.

class smooth.components.component_h2_refuel_cooling_system.**H2RefuelCoolingSystem**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the H2 refuel cooling system component
- **bus_el** (*str*) – electricity bus that is the input of the cooling system
- **nominal_value** (*numerical*) – value that the timeseries should be multiplied by, default is 1
- **csv_filename** (*str*) – csv filename containing the desired timeseries, e.g. 'my_filename.csv'
- **csv_separator** (*str*) – separator of the csv file, e.g. ',' or ';' (default is ',')
- **column_title** (*str or int*) – column title (or index) of the timeseries, default is 0
- **path** (*str*) – path where the timeseries csv file can be located
- **cool_spec_energy** (*numerical*) – energy required to cool the refuelling station [kJ/kg]
- **standby_energy** (*numerical*) – required standby energy [kJ/h]
- **life_time** (*numerical*) – life time of the component [a]
- **number_of_units** – number of units installed

- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **data** (*pandas dataframe*) – dataframe containing data from timeseries
- **electrical_energy** (*numerical*) – electrical energy required for each hour [Wh]

add_to_oemof_model (*busses*, *model*)

Creates an oemof Sink component from information given in the H2RefuelCoolingSystem class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.20 H2 CHP

This module represents another combined heat and power (CHP) system that uses hydrogen to generate electricity and heat. This module is comparable to the Fuel Cell CHP module, but using different efficiencies that are based on real life data.

2.20.1 Scope

The importance of a hydrogen CHP component in dynamic energy systems is its potential to enable better sector coupling between the electricity and heating sectors, thus less dependence on centralised power systems by offering the ability for localised energy supply [1].

2.20.2 Concept

The H2 CHP component has a hydrogen bus input and electrical and thermal bus outputs. Similarly to the fuel cell CHP component, the behaviour of the H2 CHP is non-linear and represented by oemof's Piecewise Linear Transformer component.

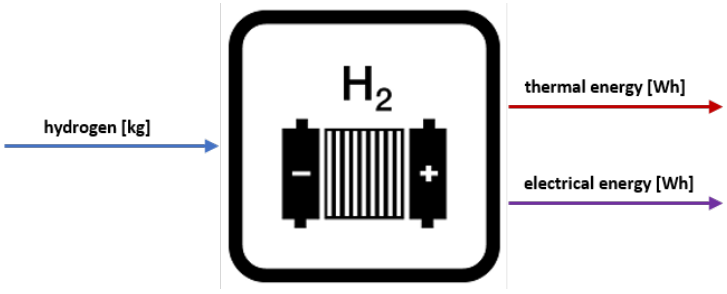

Fig. 14: Fig.1: Simple diagram of an H2 CHP.

Efficiency

The efficiency of the CHP is assumed to be constant

For more detailed information, visit *Fuel cell CHP*

class smooth.components.component_h2_chp.**H2Chp**(*params*)

Bases: *smooth.components.component.Component*

:param

add_to_oemof_model(*busses*, *model*)

Creates a non-linear oemof Transformer component to be used in the oemof model

The CHP has to be modelled as two components because the piecewise linear transformer does not accept 2 outputs yet.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns tuple of electric and thermal oemof components

get_electrical_energy_by_h2(*h2_consumption*)

get_thermal_energy_by_h2(*h2_consumption*)

update_constraints(*busses*, *model_to_solve*)

Sometimes special constraints are required for the specific components, which can be written here. Else, this function is used as placeholder for components without constraints.

Parameters

- **busses** (*dict*) – Dict of the virtual buses used in the energy system
- **model_to_solve** – ToDo: look this up in oemof

Returns If used as a placeholder, nothing will be returned. Else, refer to specific component that uses the update_constraints function for further detail.

update_flows(*results*)

Updates the flows of a component for each time step.

Parameters

- **results** (*object*) – The oemof results for the given time step
- **comp_name** (*str, optional*) – The name of the component - while components can generate more than one oemof model, they sometimes need to give a custom name, defaults to None

Returns updated flow values for each flow in the 'flows' dict

2.21 PEM Electrolyzer

Polymer electrolyte membrane (PEM) electrolyzer agents are created through this class.

2.21.1 Scope

Despite being less mature in the development phase than alkaline electrolysis, thus having higher manufacturing costs, PEM electrolysis has advantages such as quick start-up times, simple maintenance and composition, as well as no dangers of corrosion [1]. If the manufacturing costs of PEM electrolyzers can be reduced by economy of scale, these electrolyzers have the potential to be crucial components in a self-sufficient renewable energy system.

2.21.2 Concept

The PEM electrolyzer is modelled using oemof's Piecewise Linear Transformer components, and the component as a whole represents the intake of electricity to produce hydrogen and waste heat as a by-product.

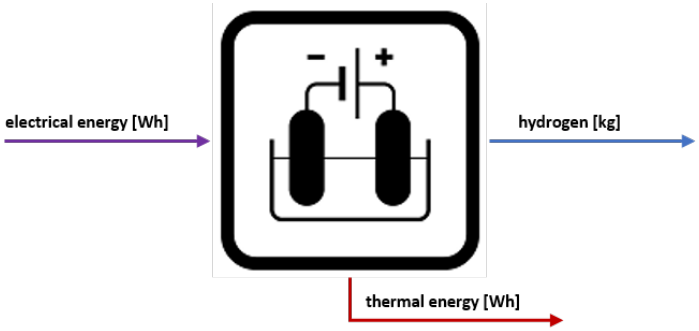

Fig. 15: Fig.1: Simple diagram of a PEM electrolyzer.

Efficiency

The amount of hydrogen and waste heat production is dependant on variable efficiencies according to nominal load, as displayed in Figure 2. The piecewise linear representation that is actually used in the SMOOTH component is shown in the left image, and the approximated efficiency curve is shown in the right image.

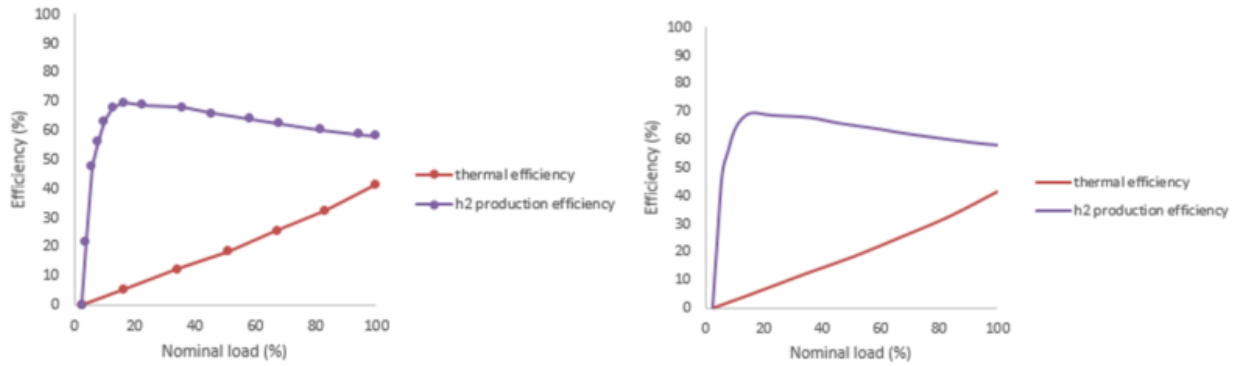

Fig. 16: Fig.2: Piecewise and approximated efficiency of PEM electrolyzer.

Hydrogen and waste heat production

The maximum power of the electrolyzer, as defined by the user, is used to determine how much electricity is consumed at each load break point.

$$bp_{el,H_2,i} = bp_{load,H_2,i} \cdot P_{max}$$
$$bp_{el,th,i} = bp_{load,th,i} \cdot P_{max}$$

- $bp_{el,H_2,i}$ = ith hydrogen break point in terms of electricity consumption [Wh]
- $bp_{load,H_2,i}$ = ith hydrogen break point in terms of nominal load [-]
- $bp_{el,th,i}$ = ith thermal break point in terms of electricity consumption [Wh]

- $bp_{load,th,i}$ = ith thermal break point in terms of nominal load [-]

From these electricity consumption values, the absolute hydrogen and waste heat energy produced at each break point is calculated:

$$H_{2_{prod,i}} = \frac{bp_{el,H_2,i} \cdot \mu_{H_2,i}}{LHV_{H_2} \cdot 1000}$$

$$E_{th,i} = bp_{el,th,i} \cdot \mu_{th,i}$$

- $H_{2_{prod,i}}$ = ith absolute hydrogen production value [kg]
- $\mu_{H_2,i}$ = ith hydrogen production efficiency [-]
- $E_{th,i}$ = ith absolute thermal energy value [Wh]
- $\mu_{th,i}$ = ith thermal efficiency [-]

Piecewise Linear Transformer

The PEM electrolyzer component uses oemof's Piecewise Linear Tansformer component in a similar fashion to the fuel cell CHP and the biogas CHP. For more detail on the usage, visit the Fuel Cell CHP or Gas Engine CHP Biogas components.

References

[1] Guo, Y. et.al. (2019). Comparison between hydrogen production by alkaline water electrolysis and hydrogen production by PEM electrolysis. IOP Conference Series: Earth and Environmental Science.

class smooth.components.component_pem_electrolyzer.**PemElectrolyzer**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to PEM electrolyzer component
- **bus_el** (*str*) – electricity bus input of the PEM electrolyzer
- **bus_h2** (*str*) – hydrogen bus output of the PEM electrolyzer
- **bus_th** (*str*) – thermal bus output of the PEM electrolyzer
- **power_max** (*numerical*) – maximum electrical input power [W]
- **life_time** (*str*) – life time of the component [a]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **heating_value** (*numerical*) – heating value of hydrogen [kWh/kg]
- **bp_load_h2_prod** (*list*) – hydrogen production efficiency load break points [-]
- **bp_eff_h2_prod** (*list*) – hydrogen production efficiency break points [-]
- **bp_load_waste_heat** (*list*) – waste heat efficiency load break points [-]
- **bp_eff_waste_heat** (*list*) – waste heat efficiency break points [-]
- **bp_elec_consumed_h2_prod** (*list*) – converted hydrogen production load points according to maximum power per timestep [Wh]
- **bp_elec_consumed_waste_heat** (*list*) – converted waste heat load points according to maximum power per timestep [Wh]

- **bp_h2_production** (*list*) – absolute hydrogen production values over the load points [kg]
- **bp_waste_heat_energy** – absolute waste heat energy values over the load points [Wh]
- **bp_elec_consumed_h2_prod_half** (*list*) – half the amount of electricity that is consumed [Wh]
- **bp_elec_consumed_waste_heat_half** – half the amount of electricity that is consumed [Wh]
- **model_h2** (*model*) – hydrogen production model to set constraints later
- **model_th** (*model*) – waste heat model to set constraints later

add_to_oemof_model(*busses*, *model*)
Creates two separate oemof Piecewise Linear Transformer components for the hydrogen and waste heat production of the PEM electrolyzer from information given in the PEMElectrolyzer class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*model*) – oemof model containing the hydrogen production and waste heat production of the PEM electrolyzer

Returns tuple of H2 and waste heat oemof components

get_h2_production_by_electricity(*electricity_consumption*)
Gets the hydrogen produced by the according electricity consumption value.

Parameters **electricity_consumption** – electrcity consumption value [Wh]

Returns according hydrogen production value [kg]

get_waste_heat_energy_by_electricity(*electricity_consumption*)
Gets the waste heat produced by the according electricity consumption value.

Parameters **electricity_consumption** – electrcity consumption value [Wh]

Returns according waste heat value [Wh]

update_constraints(*busses*, *model_to_solve*)
Set a constraint so that the electricity inflow of the hydrogen and the waste heat part are always the same (which is necessary while the piecewise linear transformer cannot have two outputs yet and therefore the two parts need to be separate components).

Parameters

- **busses** –
- **model_to_solve** –

Returns

update_flows(*results*)
Updates the flows of the electrolyzer components for each time step.

::param results: The oemof results for the given time step :type results: object :return: updated flow values for each flow in the 'flows' dict

2.22 Power Converter

This module represents a generic power converter. It can be used to model AC-DC, DC-AC, AC-AC or DC-DC converters.

2.22.1 Scope

Power converters play an important role in diverse renewable energy systems, by regulating and shaping electrical signals in the appropriate forms for other components in the system and the demands.

2.22.2 Concept

A simple power converter component is created which intakes an AC or DC electric bus and transforms it into a different AC or DC electric bus with an assumed constant efficiency. The default efficiency is taken to be 95%, as stated in [1][2] for AC-DC converters. In [3] the efficiency for a DC-AC converter is given with 99%. This value should hence be modified by the user in the model definition. The amount of electricity that can leave the converter is limited by the defined maximum power.

Fig. 17: Fig.1: Simple diagram of a power converter.

References

[1] Harrison, K.W. et. al. (2009). The Wind-to-Hydrogen Project: Operational Experience, Performance Testing, and Systems Integration, NREL. https://www.nrel.gov/docs/fy09osti/44082.pdf [2] Hayashi, Y. (2013). High Power Density Rectifier for Highly Efficient Future DC Distribution System, NTT Facilities Japan. [3] Sunny Highpower PEAK3 inverter (see manufacturer PDF)

class smooth.components.component_power_converter.**PowerConverter**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the AC-DC converter component
- **bus_input** (*str*) – electric input bus the converter is connected to
- **bus_output** (*str*) – electric output bus the converter is connected to
- **output_power_max** (*numerical*) – maximum output power [W]
- **efficiency** (*numerical*) – efficiency of the converter

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component using the information given in the PowerConverter class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.23 Sink

A generic sink component (usually for excess electricity, heat etc.) is created through this class.

2.23.1 Scope

A sink component is a virtual component that usually represents excesses in an energy system e.g. excess electricity or heat production.

2.23.2 Concept

The sink component is generic, where the input bus type and optionally the maximum input per time step are defined by the user. The default value is set to very high to represent a limitless capacity.

class smooth.components.component_sink.**Sink**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the sink component
- **input_max** (*numerical*) – maximum input per timestep of commodity e.g. for excess electricity [Wh], heat [Wh], hydrogen [kg]
- **bus_in** (*str*) – input bus of the sink component e.g. the electricity bus
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **commodity_costs** (*numerical*) – costs for the commodities e.g. [EUR/Wh], [EUR/kg] (negative costs means the system earns money when the commodity enters the sink component)

add_to_oemof_model(*busses*, *model*)

Creates an oemof Sink component from the information given in the Sink class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

2.24 Storage H2

This module represents a hydrogen storage tank.

2.24.1 Scope

Hydrogen can have a significant role in the integration of energy systems with its storage capabilities. By providing a capacity for storing excess electricity production, this can result in both the minimization of energy wasteage and smaller scale energy production systems whilst meeting the same demand. This is particularly important for cases of seasonal storage, where as an example, excess electricity production in the summer months with the lowest demands can be utilized at a later date.

2.24.2 Concept

The hydrogen storage component has a hydrogen bus input and a hydrogen bus output, which will sometimes be different from each other if another component in the system requires that the hydrogen has come directly from the storage, for instance.

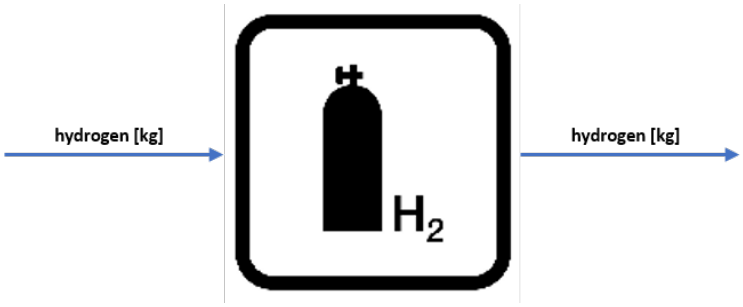

Fig. 18: Fig.1: Simple diagram of a hydrogen storage.

Initial storage level

The initial storage level is determined by the user through stating the capacity and the initial storage factor in relation to the capacity:

$$SL_{init} = F_{SL,init} \cdot C$$

- SL_{init} = initial storage level [kg]
- $F_{SL,init}$ = initial storage level as a factor of the capacity [-]
- C = storage capacity [kg]

Wanted storage level

It is optional for the user to define a wanted storage level through the storage level wanted factor (the default is set to None). If this value has been defined, then there will be different artificial costs used depending on whether the storage level is above or below the wanted level.

$$SL_{wanted} = F_{SL,wanted} \cdot C$$

- SL_{wanted} = wanted storage level [kg]
- $F_{SL,wanted}$ = wanted storage level as a factor of the capacity [-]
- C = storage capacity [kg]

Mass and volume

The minimum storage level mass at minimum pressure and the volume of the storage at maximum pressure are both calculated by initially using an iterative process where the specific volume is changed. First the initial value for the specific volume is given:

$$V_{spec,0} = 10$$

- $V_{spec,0}$ = predfined initial value for specific volume [m^3/mol]

Then using the initial value, the iterative process begins:

$$V_{spec,i+1} = \frac{R \cdot T}{p + \frac{rk_a}{T^{0.5} \cdot V_{spec,i} \cdot (V_{spec,i} + rk_b)}} + rk_b$$

- $V_{spec,i+1}$ = ith + 1 specific volume [m^3/mol]
- R = gas constant [J/(K*mol)]
- T = storage temperature [K]
- p = storage pressure [Pa] (p_{min} for calculating the mass and p_{max} for the storage)
- rk_a = Redlich Kwong equation of state parameter a
- $V_{spec,i}$ = ith specific volume [m^3/mol]
- rk_b = Redlich Kwong equation of state parameter b

After ten iterations, the specific volume value is used to obtain the storage volume and the minimum storage level mass:

$$V = C \cdot \frac{V_{spec}}{M_r}$$

$$SL_{min} = V \cdot \frac{M_r}{V_{spec}}$$

- V = storage volume [m^3]
- C = storage capacity [kg]
- V_{spec} = specific volume [m^3/mol]
- M_r = molar mass of H2 [kg/mol]
- SL_{min} = minimum storage level mass [kg]

Pressure

The pressure of the storage is calculated as follows:

$$p = \frac{R \cdot T}{(V \cdot \frac{M_r}{SL} - rk_b)} - \frac{rk_a}{T^{0.5} \cdot V \cdot \frac{M_r}{SL} \cdot (V \cdot \frac{M_r}{SL} + rk_b)}$$

- p = storage pressure [Pa]
- SL = storage level [kg]

class smooth.components.component_storage_h2.**StorageH2**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the storage component
- **bus_in** (*str*) – hydrogen bus that enters the storage
- **bus_out** (*str*) – hydrogen bus that leaves the storage
- **p_min** (*numerical*) – minimum pressure of the storage [bar]
- **p_max** (*numerical*) – maximum pressure of the storage [bar]
- **storage_capacity** (*numerical*) – storage capacity at maximum pressure (usable storage + minimum storage) [kg]
- **life_time** (*numerical*) – lifetime of the component [a]
- **initial_storage_factor** (*numerical*) – initial storage level as a factor of the capacity [-] e.g. 0.5 means half of the capacity
- **delta_max** (*numerical*) – maximum chargeable hydrogen in one time step [kg/t] where t is the step-size
- **slw_factor** (*numerical*) – storage level wanted as a factor of the capacity [-]
- **vac_in** (*numerical*) – normal variable artificial costs for charging (in) the storage [EUR/kg]
- **vac_out** (*numerical*) – normal variable artificial costs for discharging (out) the storage [EUR/kg]
- **vac_low_in** (*numerical*) – variable artificial costs for charging that apply if the storage level is below the wanted storage level [EUR/kg]
- **vac_low_out** (*numerical*) – variable artificial costs for discharging that apply if the storage level is below the wanted storage level [EUR/kg]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **storage_level_init** (*numerical*) – initial storage level [kg]
- **T_crit** (*numerical*) – critical temperature [K]
- **p_crit** (*numerical*) – critical pressure [Pa]
- **Mr** (*numerical*) – molar mass of H2 [kg/mol]
- **R** – gas constant [J/(K*mol)]
- **rk_a** (*numerical*) – Redlich Kwong equation of state parameter a
- **rk_b** (*numerical*) – Redlich Kwong equation of state parameter b
- **V** (*numerical*) – storage volume [m^3]
- **storage_level_min** (*numerical*) – mass at minimum pressure that can't be used [kg]
- **storage_level** (*numerical*) – storage level [kg]
- **pressure** (*numerical*) – storage pressure [bar]
- **current_vac** (*list*) – current artificial costs for input and output [EUR/kg]

add_to_oemof_model(*busses*, *model*)

Creates an oemof Generic Storage component from the information given in the StorageH2 class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

get_mass(*p*, *V=None*)
Calculates the mass of the storage at a certain pressure.

Parameters

- **p** (*numerical*) – pressure [bar]
- **V** (*numerical*) – storage volume [m^3]

Returns mass of the storage [kg]

get_pressure(*m*)
Calculates the storage pressure for a given mass.

Parameters **m** (*numerical*) – mass [kg]

Returns pressure [bar]

get_volume(*p*, *m*)
Calculates the volume needed to fit a certain mass at given pressure.

Parameters

- **p** (*numerical*) – pressure [bar]
- **m** (*numerical*) – mass [kg]

Returns volume of the storage [m^3]

prepare_simulation(*components*)
Prepares the simulation by setting the appropriate artificial costs and the maximum chargeable hydrogen in one time step (delta max).

Parameters **components** (*list*) – List containing each component object

Returns artificial costs and delta max

update_states(*results*)
Updates the states of the storage component for each time step

Parameters **results** (*object*) – oemof results for the given time step

Returns updated state values for each state in the 'state' dict

2.25 Stratified Thermal Storage

This module represents a stratified thermal storage tank, based on oemof thermal's component.

2.25.1 Scope

A stratified thermal storage vessel is able to store thermal energy through stratification, and thus minimise energy wastage in systems.

2.25.2 Concept

This component has been largely based on oemof thermal's stratified thermal storage component. Visit oemof thermal's readthedocs site for detailed information on how the component was constructed [1].

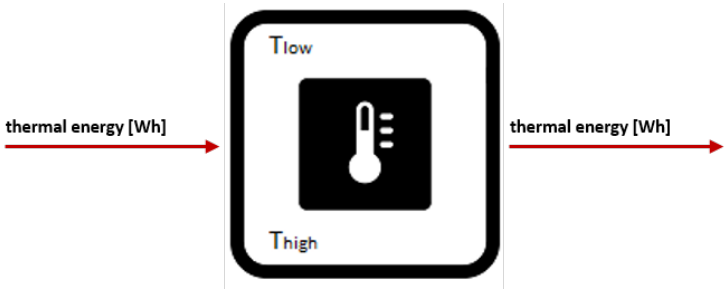

Fig. 19: Fig.1: Simple diagram of an electric heater.

References

[1] oemof thermal (2019). Stratified thermal storage, Read the Docs: https://oemof-thermal.readthedocs.io/en/latest/stratified_thermal_storage.html

class smooth.components.component_stratified_thermal_storage.**StratifiedThermalStorage**(*param*
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the stratified thermal storage component
- **bus_in** (*str*) – thermal bus input of the storage
- **bus_out** (*str*) – thermal bus output of the storage
- **storage_capacity** (*numerical*) – storage capacity [Wh]
- **storage_level_min** (*numerical*) – minimum storage level relative to storage capacity [-]
- **storage_level_max** (*numerical*) – maximum storage level relative to storage capacity [-]
- **max_heat_flow_charge** (*numerical*) – maximum heat charged into the storage per timestep [Wh]
- **max_heat_flow_discharge** (*numerical*) – maximum heat discharged from the storage per timestep [Wh]
- **storage_level_init** (*numerical*) – initial storage level [Wh]
- **life_time** (*numerical*) – lifetime of the component [a]
- **nominal_value** (*numerical*) – value that the timeseries should be multipled by, default is 1
- **csv_filename** – csv filename containing the desired demand timeseries
- **csv_separator** (*str*) – separator of the csv file e.g. ',' or ';', default is ','
- **column_title** (*str or int*) – column title (or index) of the timeseries, default is 0
- **path** (*str*) – path where the timeseries csv file can be located

- **density** (*numerical*) – density of the storage medium [kg/m3]
- **heat_capacity** (*numerical*) – heat capacity of the storage medium [J/(kg*K)]
- **temp_h** (*numerical*) – hot temperature level of the stratified storage tank [K]
- **temp_c** (*numerical*) – cold temperature level of the stratified storage tank [K]
- **temp_env** (*numerical*) – environment temperature value [C] because timeseries usually in degrees C
- **height_diameter_ratio** (*numerical*) – height to diameter ratio of storage tank [-]
- **s_iso** (*numerical*) – thickness of isolation layer [m]
- **lamb_iso** (*numerical*) – heat conductivity of isolation material [W/(m*K)]
- **alpha_inside** (*numerical*) – heat transfer coefficient inside [W/(m2*K)]
- **alpha_outside** (*numerical*) – heat transfer coefficient outside [W/(m2*K)]
- **vac_in** (*numerical*) – normal var. art. costs for charging in the storage [EUR/Wh]
- **vac_out** (*numerical*) – normal var. art. costs for discharging out the storage [EUR/Wh]
- **storage_level_wanted** (*numerical*) – if a storage level is set as wanted, the vac_low costs apply if the storage is below that level [Wh].
- **vac_low_in** (*numerical*) – var. art. costs that apply if storage level is below wanted storage level [Wh]
- **vac_low_out** (*numerical*) – var. art. costs that apply if storage level is below wanted storage level [Wh]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **storage_level** (*numerical*) – storage level [Wh]
- **current_vac** (*array*) – stores the current artificial costs for input and output [EUR/Wh]
- **volume** (*numerical*) – storage volume [m^3]
- **diameter** (*numerical*) – diameter of the storage [m]
- **u_value** (*numerical*) – thermal transmittance [W/(m2*K)]
- **loss_rate** (*numerical (sequence or scalar)*) – relative loss of the storage capacity between two consecutive timesteps [-]
- **fixed_losses_relative** (*numerical (sequence or scalar)*) – losses independent of state of charge between two consecutive timesteps relative to nominal storage capacity [-]
- **fixed_losses_absolute** (*numerical (sequence or scalar)*) – losses independent of state of charge and independent of nominal storage capacity between two consecutive timesteps [Wh]

add_to_oemof_model(*busses*, *model*)

Creates an oemof GenericStorage component from the information given in the Stratified Thermal Storage class, to be used in the oemof model

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

calculate_losses(*u_val*, *d*, *de*, *h_c*, *t_c*, *t_h*, *t_env*, *time_increment=1*)
Calculates the loss rate and the fixed losses for the stratified thermal storage

Parameters

- **u_val** (*numerical*) – thermal transmittance [W/(m2*K)]
- **d** (*numerical*) – diameter of storage tank [m]
- **de** (*numerical*) – density of the storage medium [kg/m3]
- **h_c** (*numerical*) – heat capacity of storage medium [J/(kg*K)]
- **t_c** (*numerical*) – cold temperature level of the stratified storage tank [K]
- **t_h** (*numerical*) – hot temperature level of the stratified storage tank [K]
- **t_env** (*numerical (sequence or scalar)*) – environmental temperature [K]
- **time_increment** (*numerical*) – time increment of the oemof.solph.EnergySytem [h]

Returns loss rate, relative fixed losses and absolute fixed losses

calculate_storage_u_value(*a_in*, *s_iso*, *l_iso*, *a_out*)
Calculates the u value (thermal transmittance) of storage envelope

Parameters

- **a_in** (*numerical*) – heat transfer coefficient inside [W/(m2*K)]
- **s_iso** (*numerical*) – thickness of isolation layer [m]
- **l_iso** (*numerical*) – heat conductivity of isolation material [W/(m*K)]
- **a_out** (*numerical*) – heat transfer coefficient outside [W/(m2*K)]

Returns u value

get_diameter(*V*, *h_d_ratio*)
Calculates the diameter of the storage tank

Parameters

- **V** (*numerical*) – storage tank volume [m3]
- **h_d_ratio** (*numerical*) – height to diameter ratio of storage tank [-]

Returns storage tank diameter

get_volume(*s_c*, *h_c*, *de*, *t_h*, *t_c*)
Calculates the storage tank volume

Parameters

- **s_c** (*numerical*) – storage capacity [Wh]
- **h_c** (*numerical*) – heat capacity of storage medium [J/(kg*K)]
- **de** (*numerical*) – density of the storage medium [kg/m3]
- **t_h** (*numerical*) – hot temperature level of the stratified storage tank [K]
- **t_c** (*numerical*) – cold temperature level of the stratified storage tank [K]

Returns storage tank volume

prepare_simulation(*components*)
Prepares the simulation by applying the appropriate variable artificial costs

Parameters **components** (*list*) – List containing each component object

Returns array containing var. art. costs in and out of the storage

update_states(*results*)
Updates the states of the thermal storage component for each time step

Parameters **results** (*object*) – oemof results for the given time step

Returns updated state values for each state in the 'state' dict

2.26 Supply

A generic supply component (usually for grid supplied electricity, heat etc.) is created through this class.

2.26.1 Scope

A supply component is a generic component that represents a supply to the energy system such as an electricity grid or a hydrogen grid.

2.26.2 Concept

The output bus type and the maximum output per timestep are defined by the user, and similarly to in the sink component, the default maximum output value is set to very high to represent a limitless capacity.

Artificial costs

There are some energy systems where the supply component should be incentivised to be used in certain scenarios and not in others. As an example, an energy system with renewable energy electricity production, the electricity grid as an alternative supply and hydrogen production/storage is considered.

If the hydrogen storage is over a defined threshold, then the system wants to prioritise using the stored hydrogen as an energy source instead of extracting energy from the grid, which is achieved by setting high artificial costs on the use of the grid component. If the hydrogen storage is below the defined threshold, however, then the system is incentivised to use the grid so that the storage does not entirely run out.

class smooth.components.component_supply.**Supply**(*params*)
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the supply component
- **output_max** (*numerical*) – maximum output per timestep of commodity e.g. for the electricity grid [Wh], thermal grid [Wh], H2 grid [kg/h]
- **bus_out** (*str*) – output bus of the supply component e.g. the electricity bus
- **fs_threshold** (*numerical*) – threshold value for artificial costs
- **fs_low_art_cost** (*numerical*) – low artificial cost value e.g. [EUR/Wh], [EUR/kg]
- **fs_high_art_cost** (*numerical*) – high artificial cost value e.g. [EUR/Wh], [EUR/kg]

- **fs_pressure** (*numerical*) – pressure of the supply if required (default is None) [bar]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **current_ac** (*numerical*) – current artificial cost value e.g. [EUR/Wh], [EUR/kg]

add_to_oemof_model(*busses*, *model*)
Creates an oemof Source component from the information given in the Supply class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

prepare_simulation(*components*)
Prepares the simulation by updating the artificial costs for the current time step (dependant on foreign states) and sets the total costs for the commodity for this time step (costs + artificial costs)

Parameters **components** (*list*) – List containing each component object

Returns total costs for the commodity for this time step e.g. [EUR/Wh], [EUR/kg]

2.27 Trailer Gate

A trailer gate component is created to limit the flows into the trailer component depending on whether delivery is possible or not.

2.27.1 Scope

The trailer gate component is a virtual component, so would not be found in a real life energy system, but is used in parallel with the trailer components to restrict the flows into the trailers depending on if delivery is possible or not.

2.27.2 Concept

A transformer component is used with a hydrogen bus input and a hydrogen bus output, where the hydrogen bus input comes from e.g. the production site and the hydrogen bus output goes to the trailer. The flow of hydrogen allowed to enter the trailer is controlled by the flow switch and whether delivery is possible in the given timestep or not.

class smooth.components.component_trailer_gate.**TrailerGate**(*params*)
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the trailer gate component
- **max_input** (*numerical*) – maximum mass of hydrogen that can flow into the component [kg]
- **trailer_distance** (*numerical*) – distance for trailer delivery [km]
- **driver_costs** (*numerical*) – driver costs [EUR/h]
- **bus_in** (*numerical*) – input hydrogen bus [kg]
- **bus_out** (*numerical*) – output hydrogen bus [kg]

- **round_trip_distance** (*numerical*) – round trip distance from origin to destination [km]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **flow_switch** (*int*) – determines whether there is a flow in the current timestep: 0 = off, 1 = on

add_to_oemof_model(*busses*, *model*)
Creates an oemof Transformer component from the information given in the TrailerGate class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

prepare_simulation(*components*)
Updates artificial costs for this time step (dependent on foreign states) and determines the maximum hydrogen input for the component.

Parameters **components** (*list*) – List containing each component object

Returns artificial costs and maximum allowed hydrogen input

update_var_costs()
Calculates variable costs of the component which only applies if the trailer is used, based on the distance travelled by the trailer.

2.28 Trailer Gate Cascade

A cascade trailer gate component is created to control the output flows from a trailer delivery to a destination site.

2.28.1 Scope

Similarly to the other gate components, the cascade trailer gate component is virtual and would not be found in a real life energy system. This component is used in parallel with the trailer cascade component to control how hydrogen is distributed between destination sites in the same trip.

2.28.2 Concept

A transformer component is used with a hydrogen bus input and a hydrogen bus output, where the hydrogen is inputted from the trailer and outputted to the destination site. The amount of hydrogen that can be delivered to the destination site is restricted by the maximum input value that is determined in the cascade trailer component. Notably, a different gate component should be created for each destination site.

class smooth.components.component_trailer_gate_cascade.**TrailerGateCascade**(*params*)
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the cascade trailer gate component

- **max_input** (*numerical*) – maximum mass of hydrogen that can flow into the component [kg]
- **bus_in** (*numerical*) – input hydrogen bus [kg]
- **bus_out** (*numerical*) – output hydrogen bus [kg]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)

add_to_oemof_model(*busses*, *model*)
Creates an oemof Transformer component from the information given in the TrailerGateCascade class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

prepare_simulation(*components*)
Sets the maximum input of the component by using the value calculated in the cascade trailer component as a foreign state.

Parameters **components** (*list*) – List containing each component object

Returns maximum allowed hydrogen input

2.29 Trailer H2 Delivery

This module represents a hydrogen trailer delivery from multiple production sites.

2.29.1 Scope

Hydrogen trailers can be crucial in an energy system as a means of transporting hydrogen from the production site to the destination site (e.g. a refuelling station).

2.29.2 Concept

The hydrogen trailer component is a transformer component with a hydrogen bus input and a hydrogen bus output, which should be distinct from each other in order to maintain a one way flow from the production site to the destination site.

Fig. 20: Fig.1: Simple diagram of a hydrogen delivery trailer

This component should be used in parallel with the trailer gate component. The amount of hydrogen that can be transported in a given time step is determined, and this value restricts the flow in the component. A simple depiction of how the concept for the single hydrogen delivery trailer is shown in Figure 2.

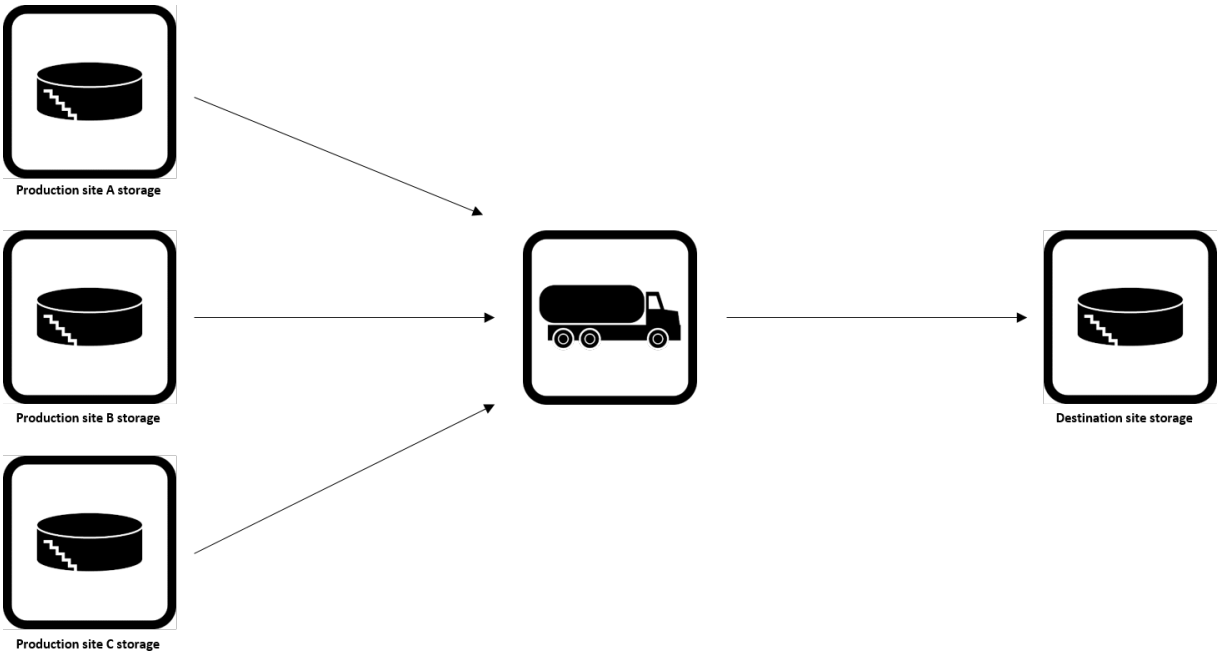

Fig. 21: Fig.2: Multiple hydrogen delivery trailer concept

Trailer activity

In this component, the trailer has the option of transporting hydrogen from multiple production sites to one destination. Thresholds are set for both the origin and destination storages. The component then:

- Checks the level of destination storage component: if it is below specified threshold, low artificial costs are implemented (to encourage system to fill it).
- Checks the level of the origin storage components and chooses the one with maximum available mass of hydrogen
- Takes into consideration the mass of hydrogen in the chosen origin storage component and the destination storage, as well as the trailer capacity, and transports the maximum possible amount of hydrogen.
- Considers the round trip distance along with the assumptions that the trailer can travel at 100 km/h and that the refuelling time for the trailer is 15 minutes. With this information, it is determined whether or not delivery is possible for the following time step with the trailer.

class smooth.components.component_trailer_h2_delivery.**TrailerH2Delivery**(*params*)
Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the trailer components
- **bus_in** (*str*) – input hydrogen bus to the trailer
- **bus_out** (*str*) – output hydrogen bus from the trailer
- **trailer_capacity** (*numerical*) – trailer capacity [kg]

- **fs_destination_storage_threshold** (*numerical*) – threshold for destination storage to encourage/discourage the use of the trailer [-]
- **hydrogen_needed** (*numerical*) – mass of hydrogen needed from delivery [kg]
- **fs_origin_available_kg** (*numerical*) – foreign state for the available mass of hydrogen in the origin storage [kg]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **current_ac** (*numerical*) – current artificial cost value [EUR/kg]

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component from the information given in the TrailerH2Delivery class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

prepare_simulation(*components*)

Prepares the simulation by determining trailer activity such as which origin storage to take from and how much hydrogen is needed.

Parameters **components** (*list*) – List containing each component object

Returns artificial costs and amount of hydrogen needed

2.30 Trailer H2 Delivery Cascade

This module represents a hydrogen trailer delivery from a single production site to one main destination site, and a secondary destination site that is dependent on the main site when necessary.

2.30.1 Scope

Hydrogen trailers can be crucial in an energy system as a means of transporting hydrogen from the production site to the destination site (e.g. a refuelling station). Sometimes it is the case that one production site should supply hydrogen to destination sites that are in close proximity to each other, for instance. If this is the case, it is beneficial for the energy system productivity to supply to both storages in one trip. This component represents a case where there is one main destination site that needs regular delivery, and one secondary destination site that will receive a delivery in the same trip as to the main destination site when necessary.

2.30.2 Concept

The cascade hydrogen trailer component is also a transformer component with a hydrogen bus input and output that are distinct from each other. This component should be used in parallel with the trailer gate and trailer gate cascade components. The amount of hydrogen that can be transported in a given time step is determined, and this value restricts the flow in the component.

A simple depiction of the concept for the single hydrogen delivery trailer is shown in Figure 2.

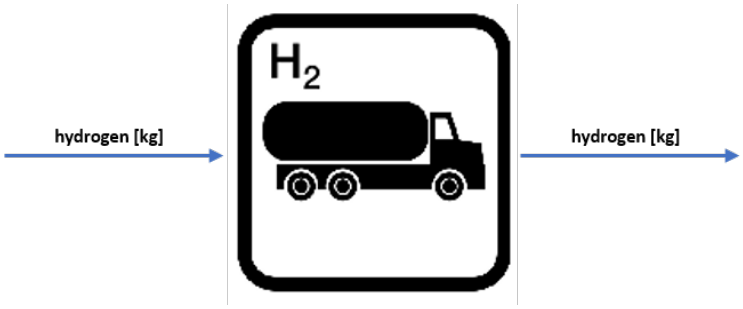

Fig. 22: Fig.1: Simple diagram of a hydrogen delivery trailer

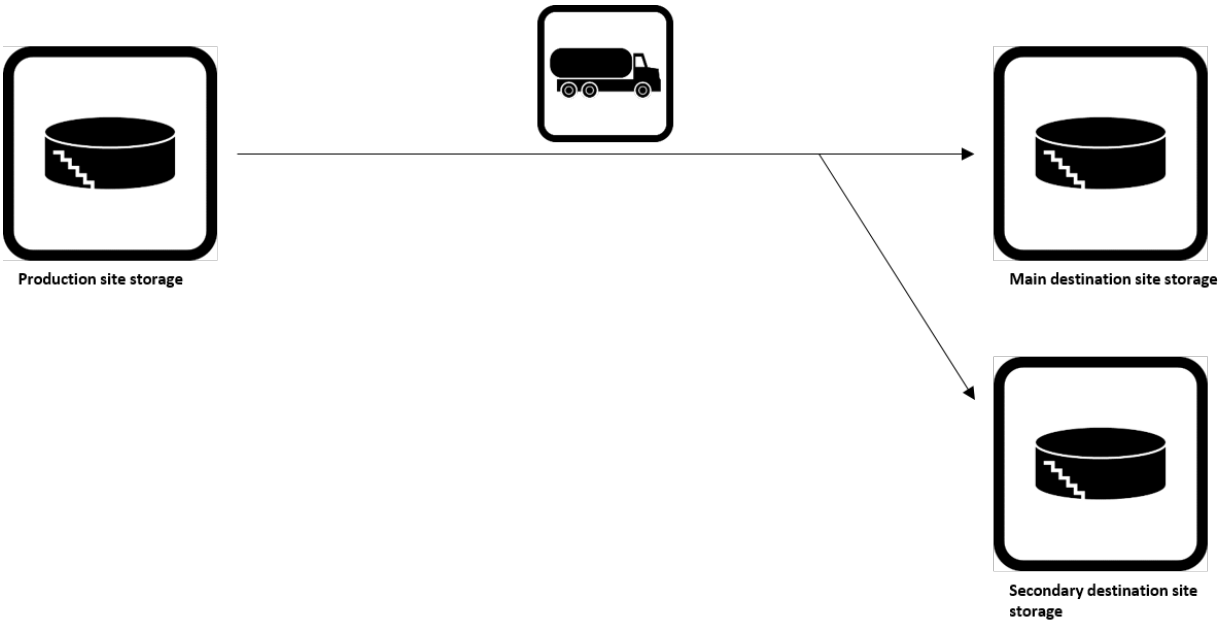

Fig. 23: Fig.2: Cascade hydrogen delivery trailer concept

Trailer activity

In this component, the trailer has the option of transporting hydrogen from one production site to two destination sites that are dependent on one another. Thresholds are set for the origin and destination storages. The component then:

- Checks the level of the origin storage component: if it is below specified threshold, the trailer cannot take hydrogen from it.
- Checks the level of destination storage components: if they are both below their specified thresholds, then the trailer is incentivised to deliver to both storages.
- Checks the mass of hydrogen in all storages as well as the trailer capacity, and transports the maximum possible amount of hydrogen.
- Calculates how much hydrogen should get delivered to the main and secondary destination storages, prioritising filling up the main storage when necessary.

class smooth.components.component_trailer_h2_delivery_cascade.**TrailerH2DeliveryCascade**(*para*

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unique name given to the trailer component
- **bus_in** (*str*) – input hydrogen bus to the trailer
- **bus_out** (*str*) – output hydrogen bus from the trailer
- **trailer_capacity** (*numerical*) – trailer capacity [kg]
- **fs_destination_storage_threshold_1** (*numerical*) – threshold for main destination storage to encourage/discourage the use of the trailer [-]
- **fs_destination_storage_threshold_2** (*numerical*) – threshold for secondary destination storage to encourage/discourage delivery to it [-]
- **hydrogen_needed** (*numerical*) – mass of hydrogen needed from delivery [kg]
- **output_h2_1** – amount of hydrogen delivered to main destination [kg]
- **output_h2_2** – amount of hydrogen delivered to secondary destination [kg]
- **fs_origin_available_kg** (*numerical*) – foreign state for the available mass of hydrogen in the origin storage [kg]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **current_ac** (*numerical*) – current artificial cost value [EUR/kg]

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component from the information given in the TrailerH2DeliveryCascade class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

prepare_simulation(*components*)

Prepares the simulation by determining trailer activity and parameters such as how much hydrogen is needed for delivery and how this should be distributed between the destination storages.

Parameters **components** (*list*) – List containing each component object

Returns artificial costs, amount of hydrogen needed and the amounts delivered to each storage

2.31 Trailer H2 Delivery Single

This module represents a hydrogen trailer delivery from a single production site.

2.31.1 Scope

Hydrogen trailers can be crucial in an energy system as a means of transporting hydrogen from the production site to the destination site (e.g. a refuelling station).

2.31.2 Concept

Similarly to the hydrogen trailer component with multiple production sites, the single hydrogen trailer component is a transformer component with a hydrogen bus input and a hydrogen bus output, which should be distinct from each other in order to maintain a one way flow from the production site to the destination site.

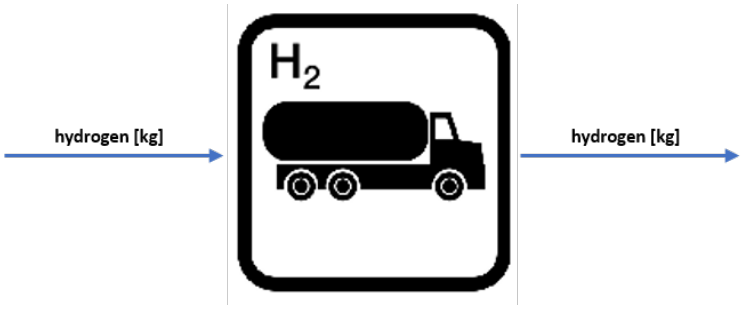

Fig. 24: Fig.1: Simple diagram of a hydrogen delivery trailer

The only difference between this component and the hydrogen trailer component is that here, there is only one option for the origin storage and this is predetermined. This component should be used in parallel with the trailer gate component. The amount of hydrogen that can be transported in a given time step is determined, and this value restricts the flow in the component. A simple depiction of how the concept for the single hydrogen delivery trailer is shown in Figure 2.

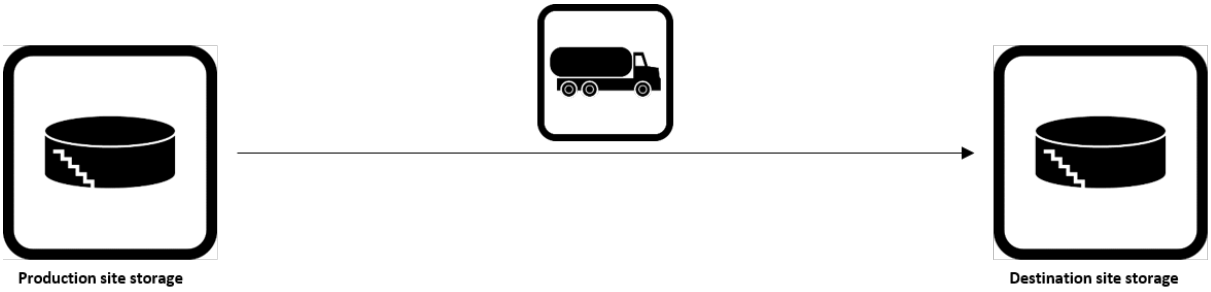

Fig. 25: Fig.2: Single hydrogen delivery trailer concept

Trailer activity

Thresholds are set for both the origin and destination storages. The component then:

- Checks the level of destination storage component: if it is below specified threshold, low artificial costs are implemented (to encourage system to fill it).
- Checks the level of origin storage component: if it is below specified threshold, the trailer cannot take any hydrogen from it.
- Checks the mass of hydrogen in both storages along with taking the trailer capacity into consideration, and transports the maximum possible amount of hydrogen.
- Considers the round trip distance along with the assumptions that the trailer can travel at 100 km/h and that the refuelling time for the trailer is 15 minutes. With this information, it is determined whether or not delivery is possible for the following time step with the trailer.

class smooth.components.component_trailer_h2_delivery_single.**TrailerH2DeliverySingle**(*params*)

Bases: *smooth.components.component.Component*

Parameters

- **name** (*str*) – unqiue name given to the trailer component
- **bus_in** (*str*) – input hydrogen bus to the trailer
- **bus_out** (*str*) – output hydrogen bus from the trailer
- **trailer_capacity** (*numerical*) – trailer capacity [kg]
- **fs_destination_storage_threshold** (*numerical*) – threshold for destination storage to encourage/discourage the use of the trailer [-]
- **hydrogen_needed** (*numerical*) – mass of hydrogen needed from delivery [kg]
- **fs_origin_available_kg** (*numerical*) – foreign state for the available mass of hydrogen in the origin storage [kg]
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **current_ac** (*numerical*) – current artificial cost value [EUR/kg]

add_to_oemof_model(*busses*, *model*)

Creates an oemof Transformer component from the information given in the TrailerH2DeliverySingle class, to be used in the oemof model.

Parameters

- **busses** (*dict*) – virtual buses used in the energy system
- **model** (*oemof model*) – current oemof model

Returns oemof component

prepare_simulation(*components*)

Prepares the simulation by determining trailer activity and how much hydrogen is needed

Parameters **components** (*list*) – List containing each component object

Returns artificial costs and the amount of hydrogen needed

2.32 Variable Grid

class smooth.components.component_var_grid.**VarGrid**(*params*)
Bases: *smooth.components.component_supply.Supply*

An electric grid with different connection levels can be created through this class

Gridlevels [1,2,3,4,5,6] describe the grid connection, if left unmodified these are: [1] house connection, [2] lov voltage grid, [3] lov voltage local network station [4] medium voltage grid, [5] medium voltage transformer station, [6] High Voltage. These can be associated with different output_max, variable costs, capex and opex for each level.

Parameters

- **self.name** (*str*) – unique name of the component
- **self.grid_level** (*int (1-6)*) – set the grid level to be used
- **self.grid_l1_output_max** (*int*) – Maximum power output of specific grid level [W]
- **self.capex_l1** (*dict*) – Capex for each grid level (e.g grid connection costs)
- **self.opex_l1** (*dict*) – Opex for each grid level (e.g grid maintanance costs)
- **self.variable_costs_l1** (*numerical*) – Variable costs for each grid level (e.g electricity costs)

2.33 External Components

This class is created for external components in the system that will not be part of the optimization, but the CAPEX and OPEX of these components should still be evaluated.

2.33.1 Scope

External components are used in the framework to represent components that do not need to be included in the simulation/optimization, but nevertheless the annuities in terms of costs and emissions for the component should be considered. The generic ExternalComponent class is the mother class for the other external components, providing a basis for what all external components must have.

2.33.2 Concept

The costs (CAPEX and OPEX) and emissions of the external component are first calculated (see the *update_financials()* and *update_emissions()* functions), and then the annuities are calculated using the *update_external_annuities()* function. It should be noted that these costs are not considered in the optimization results as they are evaluated seperately.

class smooth.components.external_component.**ExternalComponent**
Bases: object

Parameters

- **external_component** (*str*) – external component type
- **name** (*str*) – specific name of the external component (must be unique)

- **life_time** (*numerical*) – lifetime of the external component [a]
- **sim_params** (*object*) – simulation parameters such as the interval time and interest rate
- **results** (*dict*) – dictionary containing the main results for the component
- **capex** (*dict*) – capital costs
- **opex** (*dict*) – operational and maintenance costs
- **op_emissions** (*dict*) – operational emission values
- **fix_emissions** (*dict*) – fixed emission values

check_validity()
This function is called immediately after the component object is created and checks if the component attributes are valid.

Raises **ValueError** – Value error raised if the life time is not defined or is less than or equal to 0

generate_results()
Generates the results after the simulation.

Returns Results for the calculated emissions, financials and annuities

set_parameters(*params*)
Sets the parameters that have been defined by the user (in the model definition) in the necessary components, overwriting the default parameter values.

Parameters **params** (*dict ToDo: make sure of this, maybe list*) – The set of parameters defined in the specific external component class

Raises **ValueError** – raised if the parameter defined by the user is not part of the external component

Returns None

2.34 H2 Dispenser

This external component class is created to represent the dispenser unit of a hydrogen refuelling station.

2.34.1 Scope

The dispenser unit of a hydrogen refuelling station does not need to be included in the optimization of an energy system, as the number of units necessary can be calculated from the hydrogen demand, but the costs of the dispensers should be considered in the final evaluation of the energy system.

2.34.2 Concept

This component requires a demand time series in the form of a CSV file. From this file, the maximum hydrogen demand in one time step is found. Then, the number of times each hose can refuel a vehicle per hour is calculated, as well as the maximum number of vehicles that require refuelling per hour. The number of dispenser units required is then calculated as follows:

$$U = \frac{V_{max}}{H \cdot R}$$

- U = number of dispenser units required [-]

- V_{max} = maximum number of vehicles that need refuelling in an hour [-]
- H = number of hoses per dispenser [-]
- R = number of possible refuels per hour [-]

class smooth.components.external_component_h2_dispenser.**H2Dispenser**(*params*)
Bases: *smooth.components.external_component.ExternalComponent*

Parameters

- **name** (*str*) – unique name given to the H2 dispenser component
- **life_time** (*numerical*) – life time of the component [a]
- **vehicle_tank_size** (*numerical*) – vehicle tank size - 40 kg for a bus and 5 kg for a passenger car are used as default [kg]
- **number_of_hoses** (*int*) – number of hoses (access points) attached to the dispenser - default is set to 2 [-]
- **refuelling_time** (*numerical*) – refuelling time to fill up the specified tank [min] - default is set to 15
- **nominal_value** (*numerical*) – value that the timeseries should be multipled by, default is 1
- **csv_filename** – csv filename containing the desired demand timeseries e.g. 'my_demand_filename.csv'
- **csv_separator** (*str*) – separator of the csv file e.g. ',' or ';', default is ','
- **column_title** (*str or int*) – column title (or index) of the timeseries, default is 0
- **path** (*str*) – path where the timeseries csv file can be located
- **set_parameters(params)** (*function*) – updates parameter default values (see generic Component class)
- **data** (*pandas dataframe*) – dataframe containing data from timeseries
- **max_hourly_h2_demand** (*numerical*) – maximum value per timestep
- **number_of_refuels_per_hour** (*numerical*) – number of times each hose can refuel a vehicle per hour
- **max_number_of_vehicles** (*numerical*) – maximum amount of vehicles that need refuelling per hour
- **number_of_units** (*numerical*) – number of dispenser units required in order to satisfy the demand

2.35 Submodules

2.36 Module contents

CHAPTER 3

smooth.examples package

3.1 Submodules

3.2 Example Model

This example represents a simple hydrogen energy system model definition.

1. The virtual busses to be used in the system are defined as a list. In this example, an electricity bus (*bel*), a low pressure hydrogen bus (*bh2_lp*), a high pressure hydrogen bus (*bh2_hp*) and a thermal bus (*bth*) are used.

```
busses = ['bel', 'bh2_lp', 'bh2_hp', 'bth']
```

2. The components are created in a list. An example of a component being added to the list is as follows:

```
components = list()
components.append({
    'component': 'electrolyzer',
    'name': 'this_ely',
    'bus_el': 'bel',
    'bus_h2': 'bh2_lp',
    'power_max': 100e3,
    'temp_init': 293.15,
    'life_time': 20,
    'capex': {
        'key': ['free', 'spec'],
        'fitting_value': [[193, -0.366], 'cost'],
        'dependant_value': ['power_max', 'power_max']
    },
    'opex': {
        'key': 'spec',
        'fitting_value': 0.04,
        'dependant_value': 'capex',
    }
})
```

3. The simulation parameters are stated:

```
sim_params = {
    'start_date': '1/1/2019',
    'n_intervals': 10,
    'interval_time': 60,
    'interest_rate': 0.03,
    'print_progress': False,
    'show_debug_flag': False,
}
```

4. A model is created containing the above three elements

```
mymodel = {
    'busses': busses,
    'components': components,
    'sim_params': sim_params
}
```

Now this model definition is ready to be used in either a simulation or an optimization.

3.3 Example Model (costs)

This example is here to show how the various cost fitting methods can be implemented in the model definition. It should be noted that the actual cost values chosen here are arbitrary. The fitting method of the cost is chosen by the key, and the possible fitting methods are:

3.3.1 Fixed cost (*'fix'*)

Here, no fitting is done. The value given in the definition is the *cost* value. The cost value for CAPEX is taken in EUR while the cost value for OPEX is taken in EUR/a.

```
cost = cost
```

An example of this could be as follows for a compressor component:

```
components.append({
    'component': 'compressor_h2',
    'name': 'h2_compressor',
    # Busses
    'bus_h2_in': 'bh2_lp',
    'bus_h2_out': 'bh2_hp',
    # Parameters
    'bus_el': 'bel',
    'm_flow_max': 33.6 * 2,
    'life_time': 20,
    # Foreign states
    'fs_component_name': ['h2_storage', None],
    'fs_attribute_name': ['pressure', 700],
    # Financials
    'capex': {
        'key': 'fix',
        'fitting_value': None,
        'dependant_value': None,
```

(continues on next page)

(continued from previous page)

```
        'cost': 2000
    },
    'opex': {
        'key': 'fix',
        'fitting_value': None,
        'dependant_value': None,
        'cost': 200
    }
})
```

Here the cost of the compressor is independant of any other parameter (*fitting_value/dependant_value* = None), at 2000 EUR for the CAPEX and 200 EUR/a for the OPEX.

3.3.2 Specific cost (*'spec'*)

The specific cost key means that the cost is dependant on one component parameter (e.g. EUR/kW). The value of the *dependant_value* is the parameter name in the form of a string (e.g. *'power_max'*). The *fitting_value* is then multiplied with the dependant value to obtain the final costs.

```
cost = fitting_value * component[dependant_value]
```

An example of this can be seen with the following PV component:

```
components.append({
    'component': 'energy_source_from_csv',
    'name': 'pv_output',
    'bus_out': 'bel',
    'csv_filename': 'ts_pv_1kW.csv',
    'csv_separator': ',',
    'nominal_value': 100,
    'column_title': 'Power output [W]',
    'path': my_path,
    'life_time': 20,
    'capex': {
        'key': 'spec',
        'fitting_value': 975.57,
        'dependant_value': 'nominal_value',
    },
    'opex': {
        'key': 'spec',
        'fitting_value': 0.02,
        'dependant_value': 'capex',
    }
})
```

This implies that the CAPEX of the PV system is 975.57 EUR/*nominal_value* where the *nominal_value* is the number of kilowatts, and that the OPEX is 2% of the CAPEX per annum.

3.3.3 Exponential cost (*'exp'*)

The exponential fitting of the cost means that two or three entries can be given as the *fitting_value*, and the costs are then calculated in the following way:

```
for two fitting values [fv_1, fv_2]:
cost = fv_1 * exp(dependant_value * fv_2)

for three fitting values [fv_1, fv_2, fv_3]:
cost = fv_1 + fv_2 * exp(dependant_value * fv_3)
```

An example of this is shown with a wind component:

```
components.append({
    'component': 'energy_source_from_csv',
    'name': 'wind_output',
    'bus_out': 'bel',
    'csv_filename': 'ts_wind_1kW.csv',
    'csv_separator': ',',
    'nominal_value': 10,
    'column_title': 0,
    'path': my_path,
    'life_time': 10,
    'capex': {
        'key': 'exp',
        'fitting_value': [750, 0.5],
        'dependant_value': 'nominal_value',
    },
    'opex': {
        'key': 'spec',
        'fitting_value': 0.02,
        'dependant_value': 'capex',
    }
})
```

This demonstrates that the CAPEX of the wind system costs $750 \cdot e^{\frac{nv}{2}}$ EUR, and that the OPEX costs 2% of the CAPEX per annum, where *nv* is the *nominal_value*.

3.3.4 Polynomial cost (*'poly'*)

For the polynomial cost function, an arbitrary number of fitting values are defined and the cost is then calculated as follows:

```
for an arbitrary number of fitting values [fv_1, fv_2, fv_3, ..., fv_n]
cost = fv_1 + fv_2 * dependant_value**1 + fv_3 * dependant_value**2 + ...
       + fv_n * dependant_value**(n-1)
```

This can be demonstrated with the costs of a storage component:

```
components.append({
    'component': 'storage_h2',
    'name': 'h2_storage',
    'bus_in': 'bh2_lp',
    'bus_out': 'bh2_lp',
    'p_min': 5,
    'p_max': 450,
    'storage_capacity': 500,
    'life_time': 30,
    'capex': {
        'key': 'poly',
        'fitting_value': [604.6, 0.5393],
```

(continues on next page)

(continued from previous page)

```
        'dependant_value': 'p_max'
    },
    'opex': {
        'key': 'spec',
        'fitting_value': 0.01,
        'dependant_value': 'capex'
    },
})
```

Here, the costs for the storage component are $604.6 + (p_{max} \cdot 0.5393)$ for the CAPEX (EUR) and the OPEX is 1% of the CAPEX per annum.

3.3.5 Free cost (*'free'*)

The free cost is similar to the polynomial fitting, but here the exponents can be chosen freely:

```
for an even number of fitting values [fv_1, fv_2, fv_3, ..., fv_n]
cost = fv_1 * dependant_value**fv_2 + fv_3 * dependant_value**fv_4 + ...
       + fv_(n-1) * dependant_value**fv_n
```

This is also demonstrated with the storage component:

```
components.append({
    'component': 'storage_h2',
    'name': 'h2_storage',
    'bus_in': 'bh2_lp',
    'bus_out': 'bh2_lp',
    'p_min': 5,
    'p_max': 450,
    'storage_capacity': 500,
    'life_time': 30,
    'capex': {
        'key': 'free',
        'fitting_value': [600, 0.5, 0.8, 0.2],
        'dependant_value': 'p_max'
    },
    'opex': {
        'key': 'spec',
        'fitting_value': 0.01,
        'dependant_value': 'capex'
    },
})
```

This means that the CAPEX for the storage would be $600 \cdot p_{max}^{0.5} + 0.8 \cdot p_{max}^{0.2}$ (EUR) and the OPEX would be 1% of the CAPEX per annum.

3.3.6 Addition of two functions

It is also possible to add two functions together if the cost equation requires this. An example of this can again be seen in a storage component where both the specific and polynomial fittings are used:

```
components.append({
    'component': 'storage_h2',
    'name': 'h2_storage',
```

(continues on next page)

(continued from previous page)

```
    'bus_in': 'bh2_lp',
    'bus_out': 'bh2_lp',
    'p_min': 5,
    'p_max': 450,
    'storage_capacity': 500,
    'life_time': 30,
    'capex': {
        'key': ['spec', 'poly'],
        'fitting_value': [600, ['cost', 100]],
        'dependant_value': ['storage_capacity', 'p_max'],
    },
    'opex': {
        'key': 'spec',
        'fitting_value': 0.01,
        'dependant_value': 'capex'
    },
})
```

The above example entails that the CAPEX of the storage component here is $600 \cdot s_c + 100 \cdot p_{max}$. In stages it can be broken down as follows:

- The first part of the cost is calculated using the specific function ($600 \cdot s_c$).
- Then the value for this is taken as the new *'cost'* value which can be then used as a free value for further calculations.
- The previously calculated *'cost'* value is then used as the first free variable in a polynomial function to obtain $600 \cdot s_c + 100 \cdot p_{max}$.

3.3.7 Variable dicts for costs (CAPEX/OPEX)

There is also the option to include multiple dictionaries containing varying cost functions depending on a parameter. As an example, this can be useful for considering the economies of scale, where specific costs of a unit decrease with increasing scale. The variable dicts for costs can be defined as follows:

```
components.append({
    'component': 'supply',
    'name': 'from_grid',
    'bus_out': 'bel',
    'output_max': 1200e3,
    'variable_costs': 0.00001,
    'dependency_flow_costs': ('from_grid', 'bel'),
    'life_time': 1,
    'capex': {
        'key': 'variable',
        'var_dict_dependency': 'output_max',
        'var_dicts':
            [
                {
                    'low_threshold': 0,
                    'high_threshold': 900e3,
                    'key': 'free',
                    'fitting_value': [2, 3],
                    'dependant_value': 'output_max'
                },
                {
```

(continues on next page)

(continued from previous page)

```
                    'low_threshold': 1000e3,
                    'high_threshold': 5000e3,
                    'key': ['spec', 'poly'],
                    'fitting_value': [10, ['cost', 1]],
                    'dependant_value': ['output_max', 'life_time'],
                },
                {
                    'low_threshold': 5000e3,
                    'high_threshold': float('inf'),
                    'key': 'spec',
                    'fitting_value': 50,
                    'dependant_value': 'output_max',
                },
            ],
    },
    'opex': {
        'key': 'variable',
        'var_dict_dependency': 'output_max',
        'var_dicts': [
            {
                'low_threshold': 0,
                'high_threshold': 1000e3,
                'key': 'spec',
                'fitting_value': 0.04,
                'dependant_value': 'capex',
            },
            {
                'low_threshold': 1000e3,
                'high_threshold': 5000e3,
                'key': 'spec',
                'fitting_value': 0.02,
                'dependant_value': 'capex',
            },
        ]
    },
})
```

This shows the varying CAPEX and OPEX costs of the electricity supply from the grid, depending on its size. If the key 'variable' is defined, multiple CAPEX or OPEX costs can be defined depending on the value of one attribute of the component. This attribute is defined for the 'var_dict_dependency' key.

The specific dict that is used in the system is chosen if:

```
low_threshold <= value(var_capex_dependency) < high_threshold
```

It should be noted that the number of dicts can be chosen freely, but they must be defined in ascending order. Also, gaps are fine between defined ranges whereas overlapping ranges are not possible. However, any values that lie within the gaps cannot be considered in the system because a cost has not been assigned to these values. The above example states that:

- If the chosen maximum output power from the grid is less than 900 kW, the CAPEX is $2 \cdot output_{max}^3$ EUR
- If the chosen maximum output power from the grid is between 1000 kW and 5000 kW, the CAPEX is $10 \cdot output_{max} + lifetime$ EUR
- If the chosen maximum output power from the grid is above 5000 kW, the CAPEX is $50 \cdot output_{max}$ EUR
- If the chosen maximum output power from the grid is less than 1000 kW, the OPEX is 4% of the CAPEX

- If the chosen maximum output power from the grid is between 1000 kW and 5000 kW, the OPEX is 2% of the CAPEX

3.4 Example Model (dict)

This example demonstrates how the components can be created in a dictionary instead of a list, which has advantages over the list form such as the inherent uniqueness of component names as well as easier access to components by name. This is particularly useful for large systems with many components.

An example of a component created as a dictionary entry is displayed below:

```
components = {
    "this_ely": {
        "component": "electrolyzer",
        "bus_el": "bel",
        "bus_h2": "bh2_lp",
        "power_max": 100000.0,
        "temp_init": 293.15,
        "life_time": 20,
        "capex": {
            "key": [
                "free",
                "spec"
            ],
            "fitting_value": [
                [
                    193,
                    -0.366
                ],
                "cost"
            ],
            "dependant_value": [
                "power_max",
                "power_max"
            ]
        },
        "opex": {
            "key": "spec",
            "fitting_value": 0.04,
            "dependant_value": "capex"
        }
}
```

3.5 Example Model (emissions)

This example represents a simple hydrogen energy system with the inclusion of emissions as well as costs.

The only difference between this example and the Example Model is that additional parameters are added to the components with relation to emissions. For example, the electrolyzer component is now defined as follows:

```
components.append({
    'component': 'electrolyzer',
    'name': 'this_ely',
    # Busses
```

(continues on next page)

(continued from previous page)

```
    'bus_el': 'bel',
    'bus_h2': 'bh2_lp',
    # Parameters
    'power_max': 100e3,
    'temp_init': 293.15,
    'life_time': 20,
    # Foreign states
    # Financials
    'capex': {
        'key': ['free', 'spec'],
        'fitting_value': [[193, -0.366], 'cost'],
        'dependant_value': ['power_max', 'power_max']
    },
    'opex': {
        'key': 'spec',
        'fitting_value': 0.04,
        'dependant_value': 'capex',
    },
    # Emissions
    'fix_emissions': {
        'key': ['free', 'spec'],
        'fitting_value': [[193, -0.366], 'cost'],
        'dependant_value': ['power_max', 'power_max']
    },
})
```

3.6 Example Model (external components)

This example is similar to the Example Model but with the inclusion of external components that are not included in the simulation/optimization, although the costs should still be considered.

The only changes in this example are the creation of external components in a list. The 'H2 Dispenser' external component is used here, which calculates the needed number of dispenser units to fulfill the hydrogen load (specified in a given CSV file). External components can be included in the model definition in the following way:

```
external_components = list()

external_components.append({
    'external_component': 'h2_dispenser',
    'name': 'test',
    'life_time': 20,
    # Financials
    'capex': {
        'key': 'spec',
        'fitting_value': 107000,
        'dependant_value': 'number_of_units'
    },
    'opex': {
        'key': 'spec',
        'fitting_value': 0.05,
        'dependant_value': 'capex'
    },
    'csv_filename': 'ts_demand_h2.csv',
    'nominal_value': 1,
```

(continues on next page)

(continued from previous page)

```
    'column_title': 'Hydrogen load',
    'path': my_path
})
```

And now the model includes the external components too:

```
mymodel = {
    'busses': busses,
    'components': components,
    'sim_params': sim_params,
    'external_components': external_components
}
```

3.7 Example Model (plotting dicts)

In order to label the components and busses differently to those set in the components and model description, dictionaries are created which are then imported in other functions such as *plot_smooth_results()*. An English set of dictionaries and a German set of dictionaries has been created to easily switch between the two languages, and the same can be applied to any other language.

3.8 Run Optimization Example

This example demonstrates how to use the optimization algorithm.

1. Define the optimization parameters. This dict needs the following information:
   - genetic algorithm parameters
   - information on the attributes to vary
2. Define the variables for the genetic algorithm:
   - number of individuals in the population
   - number of generations that will be evaluated
   - number of cores used in the optimization
   - the visibility of the pareto front
   - the occurence of post-processing
   - the objective functions to maximise/minimise
   - whether or not intermediate results should be saved
   - whether or not the detailed final results should be saved

For instance, in the below example of the variables being defined, the optimization is based on minimizing costs and emissions, evaluating 8 individuals for 2 generations with the use of the maximum possible number of cores available.

```
opt_params['ga_params'] = {
    'population_size': 8,
    'n_generation': 2,
    'n_core': 'max',
    'plot_progress': True,
```

(continues on next page)

(continued from previous page)

```
    'post_processing': True,
    'save_intermediate_results': True,
    'objectives': (
        lambda x: -sum([c.results["annuity_total"] for c in x]),
        lambda x: -sum([c.results["annual_total_emissions"] for c in x]),
    ),
    'objective_names': ('costs', 'emissions'),
    'SAVE_ALL_SMOOTH_RESULTS': False,
}
```

3. Define the attribute variation information that will be used by the genetic algorithm:

- component name
- component attribute that will be varied in the optimization
- the range (minimum and maximum value) to be considered in the variation process
- the stepsize to be considered within the range

As an example, here is how the variation of an electrolyzer's power between 100 kW and 2000 kW with a stepsize of 50 kW is defined:

```
var_ely_power = {
    'comp_name': 'this_ely',
    'comp_attribute': 'power_max',
    'val_min': 100e3,
    'val_max': 2000e3,
    'val_step': 50e3
}
```

4. Add the attribute variation information to the optimization parameters:

```
opt_params['attribute_variation'] = [var_ely_power, var_storage_capacity]
```

3.9 Run Smooth Example

This example shows how a simulation in SMOOTH can be defined.

- The *run_smooth()* function is called which instigates the simulation, and the results are saved in the *smooth_result* parameter.
- The results are plotted using the *smooth_result* and the dictionary of choice for the axis/labels with the *plot_results()* function.
- The results are printed in the terminal by calling the *print_results()* function.
- The results are saved as a pickle file with the *save_results()* function, that can later be loaded with the *load_results()* function.
- The costs of the external components are calculated by using the *costs_for_ext_components()* function.

3.10 Module contents

CHAPTER 4

smooth.framework package

4.1 Subpackages

4.1.1 smooth.framework.functions package

Submodules

smooth.framework.functions.calculate_external_costs module

smooth.framework.functions.calculate_external_costs.**costs_for_ext_components**(*model*)
Calculates costs for components in the system which are not to be included in the optimization but their costs must still be taken into consideration. The name of an external component must be unique within the model.

Parameters **model** (*dictionary*) – smooth model

Returns external components

Return type list of *Component*

Raises **ValueError** – an external component name is not unique within the model

smooth.framework.functions.debug module

smooth.framework.functions.debug.**get_df_debug**(*df_results*, *results_dict*, *new_df_results*)
Generate debug info from results.

Parameters

- **df_results** (*pandas dataframe*) – results dataframe to compare against (e.g. last iteration)
- **results_dict** – results dictionary from oemof.processing.parameter_as_dict
- **new_df_results** (*pandas dataframe*) – newest results dataframe

Returns debug dataframe

Return type pandas dataframe

Raises **TypeError** – if df_results or results_dict is not set

smooth.framework.functions.debug.**show_debug**(*df_debug*, *components*)
Print and plot debug info, save to file

Parameters

- **df_debug** (*pandas dataframe*) – debug dataframe
- **components** (list of *Component*) – result from run_smooth for plotting

smooth.framework.functions.functions module

smooth.framework.functions.functions.**choose_valid_dict**(*component*, *var_dict*)
Function to select a valid dict (capex / fix_emissions) depending on the value of an attribute of the specific component.

todo: check opex and op_emissions

Parameters

- **component** (class:~*smooth.components.component.Component*) – object of this component
- **var_dict** (*dict*) – dict object (capex/fix_emissions) of this component

Returns Valid dictionary (capex/fix_emissions) for the actual value of the depending parameter of the component

smooth.framework.functions.functions.**create_component_obj**(*model*, *sim_params*)
Create components from model.

Parameters

- **model** (*dictionary*) – smooth model
- **sim_params** (*SimulationParameters*) – simulation parameters

Returns list of components in model

Return type list of *Component*

smooth.framework.functions.functions.**cut_suffix**(*name*, *suffix*)
Cuts off the *suffix* from *name* string, if it ends with it

Parameters

- **name** (*string*) – original name from which suffix will be cut off
- **suffix** – string to be removed

Returns string without suffix

smooth.framework.functions.functions.**cut_suffix_loop**(*name_tuple*, *suffix_list*)
Cuts off all suffixes present in *suffix_list* from names in *name_tuple*

Parameters

- **name_tuple** – tuple of strings from which suffixes will be cut off
- **suffix_list** – list of strings to be removed

Returns updated *name_tuple*

Return type tuple of strings

smooth.framework.functions.functions.**extract_flow_per_bus**(*smooth_result*, *name_label_dict*)

Extract dict containing the busses that will be plotted.

Parameters

- **smooth_result** (list of *Component*) – result from run_smooth
- **name_label_dict** – dictionary with key being a component name in the model and value the name to display

Returns dictionary of all busses from the model with their flow values over time

smooth.framework.functions.functions.**get_date_time_index**(*start_date*, *n_intervals*, *step_size*)

Function defining the parameters for perfect/myopic foresight.

Parameters

- **start_date** (*string*) – the first evaluated time period, e.g. '1/1/2019'
- **n_intervals** (*integer*) – number of time periods
- **step_size** (*number*) – length of one time step in minutes

Returns *n_intervals* dates, each *step_size* minutes apart

Return type pandas DateTimeIndex

smooth.framework.functions.functions.**get_sim_time_span**(*n_interval*, *step_size*)

Calculate the time span of the simulation.

Parameters **n_interval** (*integer*) – number of intervals

Step_size length of one time step in minutes

Returns time delta in minutes

Return type number

smooth.framework.functions.functions.**interval_time_index**(*date_time_index*, *i_interval*)

Function to divide the set date time index into hourly intervals.

This function seems to be unused.

Parameters

- **date_time_index** (*DateTimeIndex*) – chosen date range for the model
- **i_interval** (*integer*) – current interval index

Returns

pandas DateTimeIndex for current interval

smooth.framework.functions.functions.**read_data_file**(*path*, *filename*, *csv_separator*, *column_title*)

Function to read the input data files.

Parameters

- **path** (*string*) – path where the csv file is located
- **filename** (*string*) – name of csv file
- **csv_separator** (*character*) – separator of csv data
- **column_title** (*string*) – title of data column

Returns column of data from csv file

Return type pandas dataframe

smooth.framework.functions.functions.**replace_at_idx**(*tup*, *i*, *val*)
Replaces a value at index *i* of a tuple *tup* with value *val*

Parameters

- **tup** – tuple to be updated
- **i** (*integer*) – index at which the value should be replaced
- **val** (*value*) – new value at index i

Returns new tuple with replaced value

smooth.framework.functions.load_results module

smooth.framework.functions.load_results.**load_results**(*file_path*)
Load the result of either a smooth run or an optimization run by the genetic algorithm.

Parameters **file_path** (*string*) – path of the result pickle file

smooth.framework.functions.plot_results module

smooth.framework.functions.plot_results.**plot_smooth_results**(*smooth_result*, *comp_label_dict={'CHP_Methane': 'Biogas-BHKW', 'ch4_grid': 'Biogas-Zufuhr', 'el_demand': 'Strombedarf', 'from_grid': 'Strombezug', 'fuel_cell_chp': 'Brennstoffzelle', 'h2_compressor': 'Wasserstoffkompressor (Hochdruck)', 'h2_compressor_from_ely': 'Wasserstoffkompressor (Niederdruck)', 'h2_demand': 'Wasserstoffbedarf', 'h2_storage': 'Wasserstoffspeicher', 'li_battery': 'Lithium-Batterie', 'pv_output': 'PV-Anlage', 'th_demand': 'Heizbedarf', 'this_ely': 'Elektrolyseur', 'this_pem_ely': 'PEM-Elektrolyseur', 'to_grid': 'Stromeinspeisung', 'wind_output': 'WE-Anlage'}*, *bus_dict={'bch4': 'Biomethan-Fluss', 'bel': 'Elektrische Leistung', 'bel_pv': 'PV Leistung', 'bel_wind': 'Wind Leistung', 'bh2_hp': 'Wasserstoff-Fluss bei Hochdruck', 'bh2_lp': 'Wasserstoff-Fluss bei Niederdruck', 'bh2_mp': 'Wasserstoff-Fluss bei Mitteldruck', 'bth': 'Thermische Leistung'}*, *y_dict={'bch4': 'Biomethan in kg/h', 'bel': 'Leistung*

All plots are drawn in a new window.

Parameters

- **smooth_result** (list of *Component*) – result from run_smooth containing all components
- **comp_label_dict** (*dictionary, optional*) – component labels, key being the component name in the model and value the name to display. Defaults to comp_dict_german from example_plotting_dicts.
- **bus_dict** (*dictionary, optional*) – bus labels, key being the bus name in the model and value the name to display. Defaults to bus_dict_german from example_plotting_dicts.
- **y_dict** (*dictionary, optional*) – labels for y-axes, key being the bus names from the model to plot and value the y-axis labels. Defaults to y_dict_german from example_plotting_dicts.

smooth.framework.functions.print_results module

smooth.framework.functions.print_results.**print_smooth_results**(*smooth_results*)
Print the financial results of a smooth run.

Parameters **smooth_results** (list of *Component*) – result from run_smooth containing all components

smooth.framework.functions.save_results module

smooth.framework.functions.save_results.**save_results**(*file_name*, *result_data*)
Save the result of either a smooth run or an optimization run by the genetic algorithm.

Parameters

- **file_name** (*string*) – name of the result pickle file
- **result_data** – data to save

smooth.framework.functions.update_annuities module

smooth.framework.functions.update_annuities.**calc_annual_emissions**(*component*, *target*)
Calculate annual emissions.

Parameters

- **component** (*Component*) – object of this component
- **target** (*dict*) – dictionary with *cost* key, e.g. component.fix_emissions

Returns annual emissions of target [kg/a]

Return type number

smooth.framework.functions.update_annuities.**calc_annuity**(*component*, *target*)
Calculate annuity

Parameters

- **component** (*Component*) – object of this component

- **target** (*dict*) – dictionary with *cost* key, e.g. component.capex

Returns annuity of target [EUR/a]

Return type number

smooth.framework.functions.update_annuities.**update_annuities**(*component*)
Compute the annual CAPEX, variable costs and emissions.

Annuities are written into the *results* dictionary of the component.

Parameters **component** (*Component*) – object of this component

smooth.framework.functions.update_annuities.**update_external_annuities**(*component*)
Convert the CAPEX to annuities

Annuities are written into the *results* dictionary of the component.

Parameters **component** (*Component*) – object of this component

smooth.framework.functions.update_fitted_cost module

smooth.framework.functions.update_fitted_cost.**get_dependant_value**(*component*, *fitting_dict*, *index*, *fixedCost*)
Get an attribute of the component as the dependant value.

Parameters

- **component** (*Component*) – object of this component
- **fitting_dict** (*dict*) – usually financial or emission object of this component
- **index** (*integer*) – current position in fitting_dict
- **fixedCost** (*string*) – key of fixed type in fitting_dict

Returns calculated costs using exponential fitting

Return type number or None

smooth.framework.functions.update_fitted_cost.**get_exp**(*component*, *fitting_dict*, *index*, *dependant_value*)
Case: An exponential fitting of the cost function is wanted.

Here 3 variables are used in the following order:

```
# for 2 fitting parameters
fv_1*exp(fv_2)

# for 3 fitting parameters
fv_1 + fv_2*exp(fv_3*Parameter)
```

Parameters

- **component** (*Component*) – object of this component
- **fitting_dict** (*dict*) – usually financial or emission object of this component
- **index** (*integer*) – current position in fitting_dict
- **dependant_value** (*number*) – dependent attribute value of object

Returns calculated costs using exponential fitting

Return type number

smooth.framework.functions.update_fitted_cost.**get_free**(*component*, *fitting_dict*, *index*, *dependant_value*)

Case: A "free" fitting of the cost function is wanted.

In this case, an arbitrary number of fitting parameters can be given. They will be used in the following order: fv_1, fv_2, fv_3, ... fv_n.

Function:

```
fv_1*dependant_value^fv_2 + fv_3*dependant_value^fv_4 + ... fv_(n-1)*dependant_
↪value^fv_n
```

Parameters

- **component** (*Component*) – object of this component
- **fitting_dict** (*dict*) – usually financial or emission object of this component
- **index** (*integer*) – current position in fitting_dict
- **dependant_value** (*number*) – dependent attribute value of object

Returns calculated costs using "free" fitting

Return type number

Raises **ValueError** – if number of fitting values is odd

smooth.framework.functions.update_fitted_cost.**get_poly**(*component*, *fitting_dict*, *index*, *dependant_value*)

Case: A polynomial fitting of the cost function is wanted.

In this case, an arbitrary number of fitting parameters can be given. They will be used in the following order: fv_1, fv_2, fv_3, ... fv_n.

Function:

```
fv_1 + fv_2*dependant_value + fv_3*dependant_value^2 + ... fv_n*dependant_value^
↪(n-1)
```

It is possible to use the polynomial function to add different cost equations together. This is achieved because the result of the initial equation used (which can be any of the cost functions) is stored as the 'cost' variable, which can then be used in a following polynomial function but now as a new free variable. See the example_model_costs example for an applied case.

Parameters

- **component** (*Component*) – object of this component
- **fitting_dict** (*dict*) – usually financial or emission object of this component
- **index** (*integer*) – current position in fitting_dict
- **dependant_value** (*number*) – dependent attribute value of object

Returns calculated costs using polynomial fitting

Return type number

smooth.framework.functions.update_fitted_cost.**get_spec**(*component*, *fitting_dict*, *index*, *dependant_value*)

Case: The fitting value is multiplied with the dependant value to get the costs.

Parameters

- **component** (*Component*) – object of this component
- **fitting_dict** (*dict*) – usually financial or emission object of this component
- **index** (*integer*) – current position in fitting_dict
- **dependant_value** (*number*) – dependent attribute value of object

Returns calculated costs using a fitting value

Return type number

smooth.framework.functions.update_fitted_cost.**update_cost**(*component*, *fitting_dict*, *index*, *dependant_value*, *name*)

Update cost of component.

Parameters

- **component** (*Component*) – object of this component
- **fitting_dict** (*dict*) – usually financial or emission object of this component
- **index** (*integer*) – current position in fitting_dict
- **dependant_value** (*number*) – dependent attribute value of object
- **name** (*string*) – human readable representation of attribute to be updated, e.g. "CAPEX/OPEX" or "emissions"

Raises **ValueError** – on unknown fitting key

smooth.framework.functions.update_fitted_cost.**update_emissions**(*component*, *emissions*)

Calculate fixed and operational emissions for this component.

This function is calculating a fix and operational value for components where "fix_emissions" or "op_emissions" are dependant on certain values. The following list shows possible fitting methods. The fitting method is chosen by the "key" value given in the "emissions" dictionary:

- "fix" –> already the fix value, nothing has to be done
- "spec" –> cost value needs to be multiplied with the dependant value
- "exp" –> exponential cost fitting
- "poly" –> polynomial cost fitting
- "free" –> polynomial cost fitting with free choosable exponents
- "variable" –> definition of multiple "fix_emissions" or "op_emissions" structures: If the emission structure changes over the size of a specific value of the component, for example because of the effects of economics of scale, the special key "variable" can be used to define multiple "fix_emissions" or "op_emissions" dicts for different ranges of this value

If multiple keys are defined, the calculations are done sequentially in order.

Parameters

- **component** (component: *Component*) – object of this component
- **emissions** (*fix_emissions or op_emissions dict*) – emission object of this component

smooth.framework.functions.update_fitted_cost.**update_financials**(*component*, *financials*)

Calculate "OPEX" or "CAPEX" for this component.

This function is calculating a fix "CAPEX" and "OPEX" value for components where "CAPEX" and "OPEX" are dependant on certain values. The following list shows possible fitting methods. The fitting method is chosen by the "CAPEX" and "OPEX" key:

- "fix" –> already the fix value, nothing has to be done
- "spec" –> cost value needs to be multiplied with the dependant value
- "exp" –> exponential cost fitting
- "poly" –> polynomial cost fitting
- "free" –> polynomial cost fitting with free choosable exponents
- "variable" –> definition of multiple "CAPEX" or "OPEX" structures: If the cost structure changes over the size of a specific value of the component, for example because of the effects of economics of scale, the special key "variable" can be used to define multiple "CAPEX" or "OPEX" dicts for different ranges of this value

If multiple keys are defined, the calculations are done sequentially in order.

Parameters

- **component** (*Component*) – object of this component
- **financials** (*capex or opex dict*) – financial object of this component

Module contents

4.2 Submodules

4.3 Run SMOOTH

This is the core of smooth. It solves (M)ILP of an energy system model for discrete time steps using the Open Energy Modelling Framework solver (oemof-solph).

4.3.1 How to use

The *run_smooth()* function expects an energy model. Such a model consists of:

- energy sources
- energy sinks
- energy transformers
- buses to transport energy

Additionally, simulation parameters are needed to run the model. A model is therefore defined as a dictionary containing all *components*, buses (grouped as *busses*) and simulation parameters (grouped as *sim_params*, see *smooth.framework.simulation_parameters.SimulationParameters*).

Example:

```
{
    components: {
        name_of_first_component: {
            component: ...,
            capex: ...,
            opex: ...,
            ...
        },
        ...
    },
    busses: [
        name_of_first_bus,
        name_of_second_bus,
        ...
    ],
    sim_params: {
        start_date: ...,
        n_intervals: ...,
        interval_time: ...,
        interest_rate: 0.03,
        ...
    }
}
```

Note: Legacy models (version < 0.2.0) define their components as a list with an extra field *name* for each component. This is deprecated.

4.3.2 Result

Two items are returned. The second is a string describing the oemof solver return status. You want this to be 'ok', although other values are possible. The first item returned is a list of all components, each updated with

- sim_params: the original simulation parameters, plus *date_time_index* for each time step and *sim_time_span* in minutes
- results: results from the simulation
  - variable_costs*
  - art_costs*
  - variable_emissions*
  - annuity_capex
  - annuity_opex
  - annuity_variable_costs
  - annuity_total
  - annual_fix_emissions
  - annual_op_emissions
  - annual_variable_emissions
  - annual_total_emissions

- states: dictionary with component-specific attributes. Each entry is a list with values for each time step
- flows: dictionary with each flow of this component. Key is tuple (from, to), entry is list with value for each time step
- data: pandas dataframe
- (component-specific attributes)

* a list with a value for each time step

4.3.3 Implementation

The concept of *run_smooth()* is demonstrated in the figure below:

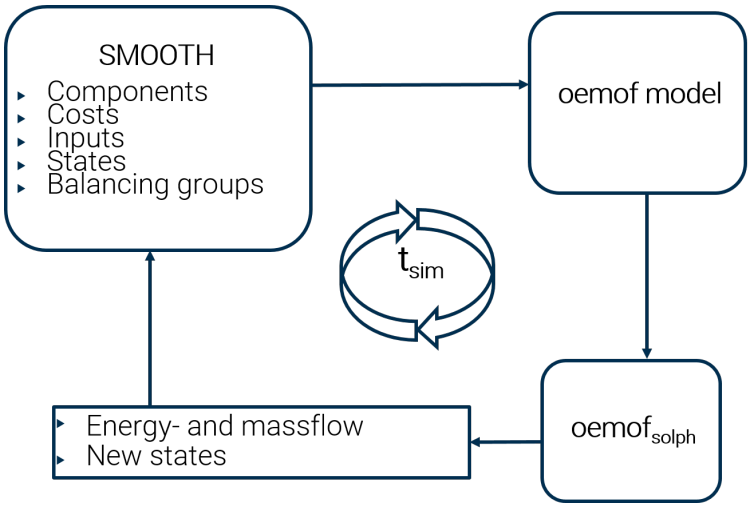

Fig. 1: Fig.1: Concept of run_smooth function.

The *run_smooth()* function has three distinct phases: initialization, simulation and post processing.

Initialization

There is not much to see here. Mainly, component instances get created from the model description. For legacy models (version < 0.2.0), the component list is converted to a dictionary. No oemof model is built here.

Simulation

This is the main part of the function. For each time step, an oemof model is solved and evaluated:

1. print current time step to console if *print_progress* is set in parameters
2. initialize oemof energy system model
3. create buses
4. update components and add them to the oemof model
5. update bus constraints
6. write lp file in current directory
7. call solver for model

8. check returned status for non#.optimal solution
9. handle results for each component
    1. update flows
    2. update states
    3. update costs
    4. update emissions

Post-processing

After all time steps have been computed, call the *generate_results* function of each component. Finally, return the updated components and the last oemof status.

smooth.framework.run_smooth.**run_smooth**(*model*)
Runs the smooth simulation framework

Parameters **model** (*dictionary*) – smooth model object containing parameters for components, simulation and busses

Returns results of all components and oemof status

Return type tuple of components and string

Raises *SolverNonOptimalError* if oemof result is not ok and not optimal

4.4 Simulation Parameters

class smooth.framework.simulation_parameters.**SimulationParameters**(*params*)
Bases: object

Class to store parameters for smooth simulation.

Parameters

- **start_date** (*string representation of date*) – the first evaluated time period. Defaults to '1/1/2019'
- **n_intervals** (*integer*) – number of time steps. Defaults to 24*7=168
- **interval_time** (*integer*) – length of one time step in minutes. Defaults to 60 (one hour)
- **interest_rate** (*float*) – Interest rate for calculating annuity out of CAPEX. Defaults to 0.03 (3%)
- **print_progress** (*boolean*) – Decide if the running progress should be printed out. Defaults to False
- **show_debug_flag** (*boolean*) – Decide if last result values should be shown in case solver was not successful. Defaults to True

Variables

- **date_time_index** – pandas date range of all time periods to be evaluated
- **sim_time_span** – length of simulation time range in minutes

set_parameters(*params*)
Helper function to set simulation parameters on initialisation.

Parameters **params** (*dictionary*) – parameters to set

Raises *ValueError* for unsupported simulation parameters

4.5 Module contents

CHAPTER 5

smooth.optimization package

5.1 Subpackages

5.2 Submodules

5.3 Run Optimization

This is the core of the genetic algorithm (GA) used for optimization. It uses the NSGA-II algorithm for multi-objective optimization of smooth components.

5.3.1 How to use

To use, call run_optimization with a configuration dictionary and your smooth model. You will receive a list of *Individual* in return. These individuals are pareto-optimal in regard to the given objective functions (limited to two functions).

An example configuration can be seen in run_optimization_example in the examples directory.

Objective functions

You may specify your custom objective functions for optimization. These should be lambdas that take the result from run_smooth and return a value. Keep in mind that this algorithm always tries to maximize. In order to minimize a value, return the negative value.

Example 1: maximize *power_max* of the first component:

```
lambda x: x[0].power_max
```

Example 2: minimize the annual costs:

```
lambda x: -sum([component.results['annuity_total'] for component in x])
```

Result

After the given number of generations or aborting, the result is printed to the terminal. All individuals currently on the pareto front are returned in a list. Their *values* member contain the component attribute values in the order given by the *attribute_variation* dictionary from the optimization params. In addition, when *SAVE_ALL_SMOOTH_RESULTS* is set to True, the *smooth_result* member of each individual contains the value returned by run_smooth.

Warning: Using SAVE_ALL_SMOOTH_RESULTS and writing the result to a file will generally lead to a large file size.

5.3.2 Implementation

Like any GA, this implementation simulates a population which converges to an optimal solution over multiple generations. As there are multiple objectives, the solution takes the form of a pareto-front, where no solution is dominated by another while maintaining distance to each other. We take care to compute each individual configuration only once. The normal phases of a GA still apply:

- selection
- crossover
- mutation

Population initialisation

At the start, a population is generated. The size of the population must be declared (*population_size*). Each component attribute to be varied in the smooth_model corresponds to a gene in an individual. The genes are initialized randomly with a uniform distribution between the minimum and maximum value of its component attribute. These values may adhere to a step size (*val_step* in `AttributeVariation`).

Selection

We compute the fitness of all individuals in parallel. You must set *n_core* to specify how many threads should be active at the same time. This can be either a number or 'max' to use all virtual cores on your machine. The fitness evaluation follows these steps:

1. change your smooth model according to the individual's component attribute values
2. run smooth
3. on success, compute the objective functions using the smooth result. These are the fitness values. On failure, print the error
4. update the master individual on the main thread with the fitness values
5. update the reference in the dictionary containing all evaluated individuals

After all individuals in the current generation have been evaluated, they are sorted into tiers by NSGA-II fast non-dominated sorting algorithm. Only individuals on the pareto front are retained, depending on their distance to their neighbors. The parent individuals stay in the population, so they can appear in the pareto front again.

Crossover

These individuals form the base of the next generation, they are parents. For each child in the next generation, genes from two randomly selected parents are taken (uniform crossover of independent genes).

Mutation

After crossover, each child has a random number of genes mutated. The mutated value is around the original value, taken from a normal distribution. Special care must be taken to stay within the component atrribute's range and to adhere to a strict step size.

After crossover and mutation, we check that this individual's gene sequence has not been encountered before (as this would not lead to new information and waste computing time). Only then is it admitted into the new generation.

Special cases

We impose an upper limit of 1000 * *population_size* on the number of tries to find new children. This counter is reset for each generation. If it is exceeded and no new gene sequences have been found, the algorithm aborts and returns the current result.

In case no individuals have a valid smooth result, an entirely new population is generated. No plot will be shown. If only one individual is valid, the population is filled up with random individuals.

Gradient ascent

The solutions of the GA are pareto-optimal, but may not be at a local optimum. Although new configurations to be evaluated are searched near the current ones, it is not guaranteed to find slight improvements. This is especially true if there are many dimensions to search and the change is in only one dimension. The chance to happen upon this single improvement is in inverse proportion to the number of attribute variations.

Therefore, the *post_processing* option exists to follow the fitness gradient for each solution after the GA has finished. We assume that each attribute is independent of each other. All solutions improve the same attribute at the same time. The number of fitness evaluations may exceed the *population_size*, however, the maximum number of cores used stays the same as before.

To find the local optimum of a single attribute of a solution, we first have to find the gradient. This is done by going one *val_step* in positive and negative direction. These new children are then evaluated. Depending on the domination, the gradient may be +*val_step*, -*val_step* or 0 (parent is optimal). Then, this gradient is followed until the child shows no improvement. The population may be topped up with multiples of *val_step* to better utilize all cores and speed up the gradient ascent. After all solutions have found their optimum for this attribute, the next attribute is varied.

Plotting

To visualize the current progress, you can set the *plot_progress* simulation parameter to True. This will show the current pareto front in a pyplot window. You can mouse over the points to show the configuration and objective values. To keep the computation running in the background (non-blocking plots) while listening for user events, the plotting runs in its own process.

On initialisation, a one-directional pipe is established to send data from the main computation to the plotting process. The process is started right at the end of the initialisation. It needs the attribute variations and objective names for hover info and axes labels. It also generates a multiprocessing event which checks if the process shall be stopped.

In the main loop of the process, the pipe is checked for any new data. This incorporates a timeout to avoid high processor usage. If new data is available, the old plot is cleared (along with any annotations, labels and titles) and

redrawn from scratch. In any case, the window listens for a short time for user input events like mouseover. Window close is a special event which stops the process, but not the computation (as this runs in the separate main process).

When hovering with the mouse pointer over a point in the pareto front, an annotation is built with the info of the *Individual*. The annotation is removed when leaving the point. A simple example of how this looks is illustrated in Figure 1. In this example, after the first generation there is one optimal energy system found which costs 244,416.21 EUR and produces 0 emissions.

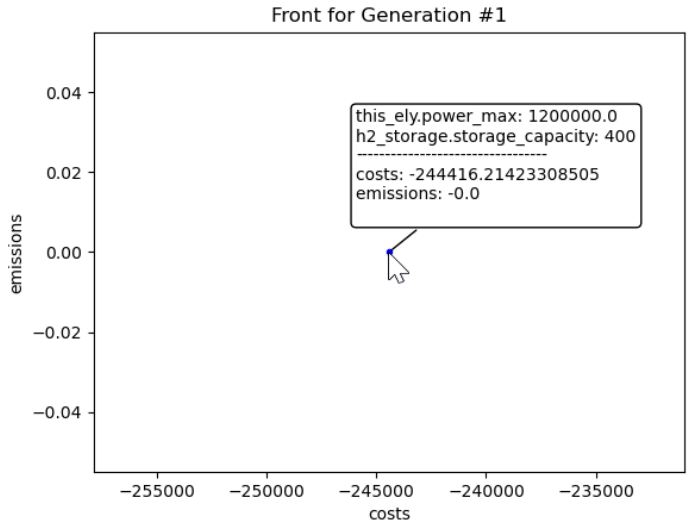

Fig. 1: Fig.1: Simple diagram of a pareto front with annotations

Sending None through the pipe makes the process show the plot until the user closes it. This blocks the process, so no new data is received, but user events are still processed.

class smooth.optimization.run_optimization.**AttributeVariation**(*iterable=()*, ***kwargs*)

Bases: object

Class that contains all information about an attribute that is varied by the genetic algorithm

Parameters

- **comp_name** (*string*) – name of component that gets varied
- **comp_attribute** (*string*) – component attribute that gets varied
- **val_min** (*number*) – minimum value of component attribute
- **val_max** (*number*) – maximum value of component attribute (inclusive)
- **val_step** (*number, optional*) – step size of component attribute

Variables **num_steps** – number of steps if *val_step* is set and not zero

Raises AssertionError when any non-optional parameter is missing or *val_step* is negative

class smooth.optimization.run_optimization.**Individual**(*values*)

Bases: object

Class for individuals evaluated by the genetic algorithm

Parameters **values** (*list*) – attribute values (individual configuration)

Variables

- **values** – given values
- **fitness** – fitness values depending on objective functions
- **smooth_result** – result from *run_smooth*

class IndividualIterator(*individual*)
Bases: object

Class to iterate over gene values.

fitness = None

smooth_result = None

values = None

smooth.optimization.run_optimization.**crossover**(*parent1*, *parent2*)

Uniform crossover between two parents Selects random (independent) genes from one parent or the other

Parameters

- **parent1** (*Individual*) – First parent
- **parent2** (*Individual*) – Second parent

Returns Crossover between parents

Return type *Individual*

smooth.optimization.run_optimization.**mutate**(*parent*, *attribute_variation*)
Mutate a random number of parent genes around original value, within variation

Parameters

- **parent** (*Individual*) – parent individual
- **attribute_variation** (list of *AttributeVariation*) – AV for all genes in parent

Returns child with some parent genes randomly mutated

Return type *Individual*

smooth.optimization.run_optimization.**fitness_function**(*index*, *individual*, *model*, *attribute_variation*, *dill_objectives*, *ignore_zero=False*, *save_results=False*)

Compute fitness for one individual Called async: copies of individual and model given

Parameters

- **index** (*int*) – index within population
- **individual** (*Individual*) – individual to evaluate
- **model** (*dict*) – smooth model
- **attribute_variation** (list of *AttributeVariation*) – attribute variations
- **dill_objectives** (*tuple of lambda-functions pickled with dill*) – objective functions
- **ignore_zero** (*boolean*) – ignore components with an attribute value of zero
- **save_results** (*boolean*) – save smooth result in individual?

Returns index, modified individual with fitness (None if failed) and smooth_result (none if not save_results) set

Return type tuple(int, *Individual*)

class smooth.optimization.run_optimization.**PlottingProcess**
Bases: multiprocessing.context.Process

Process for plotting the intermediate results

Data is sent through (onedirectional) pipe. It should be a dictionary containing "values" (array of *Individual*) and "gen" (current generation number, displayed in title). Send None to stop listening for new data and block the Process by showing the plot. After the user closes the plot, the process returns and can be joined.

Parameters

- **pipe** (multiprocessing pipe) – data transfer channel
- **attribute_variation** (list of *AttributeVariation*) – AV of *Optimization*
- **objective_names** (*list of strings*) – descriptive names of *Optimization* objectives

Variables

- **exit_flag** – Multiprocessing event signalling process should be stopped
- **fig** – figure for plotting
- **ax** – current graphic axis for plotting
- **points** – plotted results or None
- **annot** – current annotation or None

main()
Main plotting thread

Loops while exit_flag is not set and user has not closed window. Checks periodically for new data to be displayed.

handle_close(*event*)
Called when user closes window

Signal main loop that process should be stopped.

hover(*event*)
Called when user hovers over plot.

Checks if user hovers over point. If so, delete old annotation and create new one with relevant info from all Indivdiuals corresponding to this point. If user does not hover over point, remove annotation, if any.

class smooth.optimization.run_optimization.**Optimization**(*iterable=()*, ***kwargs*)
Bases: object

Main optimization class to save GA parameters

Parameters

- **n_core** (*int or 'max'*) – number of threads to use. May be 'max' to use all (virtual) cores
- **n_generation** (*int*) – number of generation to run

- **population_size** (*int*) – number of new children per generation. The actual size of the population may be higher - however, each individual is only evaluated once
- **attribute_variation** (list of dicts, see *AttributeVariation*) – attribute variation information that will be used by the GA
- **model** (*dict*) – smooth model
- **objectives** (*2-tuple of lambda functions*) – multi-objectives to optimize. These functions take the result from *run_smooth* and return a float. Positive sign maximizes, negative sign minimizes. Defaults to minimizing annual costs and emissions
- **objective_names** (*2-tuple of strings, optional*) – descriptive names for optimization functions. Defaults to ('costs', 'emissions')
- **post_processing** (*boolean, optional*) – improve GA solution with gradient ascent. Defaults to False
- **plot_progress** (*boolean, optional*) – plot current pareto front. Defaults to False
- **ignore_zero** (*boolean, optional*) – ignore components with an attribute value of zero. Defaults to False
- **save_intermediate_results** (*boolean, optional*) – write intermediate results to pickle file. Only the two most recent results are saved. Defaults to False
- **SAVE_ALL_SMOOTH_RESULTS** (*boolean, optional*) – save return value of *run_smooth* for all evaluated individuals. **Warning!** When writing the result to file, this may greatly increase the file size. Defaults to False

Variables

- **population** – current individuals
- **evaluated** – keeps track of evaluated individuals to avoid double computation
- **ax** – current figure handle for plotting

Raises *AttributeError* or *AssertionError* when required argument is missing or wrong

err_callback(*err_msg*)
Async error callback

Parameters **err_msg** (*string*) – error message to print

set_fitness(*result*)
Async success callback Update master individual in population and *evaluated* dictionary

Parameters **result** (tuple(index, *Individual*)) – result from fitness_function

compute_fitness()
Compute fitness of every individual in *population* with *n_core* worker threads. Remove invalid individuals from *population*

save_intermediate_result(*result*)
Dump result into pickle file in current working directory. Same content as smooth.save_results. The naming schema follows *date-time*-intermediate_result.pickle. Removes second-to-last pickle file from same run.

Parameters **result** (list of *Individual*) – the current results to be saved

gradient_ascent(*result*)
Try to fine-tune result(s) with gradient ascent

Attributes are assumed to be independent and varied separately. Solutions with the same fitness are ignored.

Parameters **result** (list of *Individual*) – result from GA

Returns improved result

Return type list of *Individual*

run()
Main GA function

Returns pareto-optimal configurations

Return type list of *Individual*

smooth.optimization.run_optimization.**run_optimization**(*opt_config*, *_model*)
Entry point for genetic algorithm

Parameters

- **opt_config** (*dict*) – Optimization parameters. May have separate *ga_params* dictionary or define parameters directly. See *Optimization*.
- **_model** (*dict or list (legacy)*) – smooth model

Returns pareto-optimal configurations

Return type list of *Individual*

5.4 Module contents

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

smooth.components, 68
smooth.components.component, 7
smooth.components.component_air_source_heat_pump, 9
smooth.components.component_battery, 11
smooth.components.component_biogas_converter, 14
smooth.components.component_biogas_smr_psa, 15
smooth.components.component_compressor_h2, 18
smooth.components.component_electric_heater, 20
smooth.components.component_electrolyzer, 22
smooth.components.component_electrolyzer_waste_heat, 25
smooth.components.component_energy_demand_from_csv, 30
smooth.components.component_energy_source_from_csv, 31
smooth.components.component_fuel_cell_chp, 32
smooth.components.component_gas_engine_chp_biogas, 36
smooth.components.component_gate, 40
smooth.components.component_h2_chp, 42
smooth.components.component_h2_refuel_cooling_system, 40
smooth.components.component_pem_electrolyzer, 43
smooth.components.component_power_converter, 47
smooth.components.component_sink, 48
smooth.components.component_storage_h2, 48
smooth.components.component_stratified_thermal_storage, 52
smooth.components.component_supply, 56
smooth.components.component_trailer_gate, 57
smooth.components.component_trailer_gate_cascade, 58
smooth.components.component_trailer_h2_delivery, 59
smooth.components.component_trailer_h2_delivery_cas 61
smooth.components.component_trailer_h2_delivery_sir 64
smooth.components.component_var_grid, 66
smooth.components.external_component, 66
smooth.components.external_component_h2_dispenser, 67
smooth.examples, 79
smooth.examples.example_model, 69
smooth.examples.example_model_costs, 70
smooth.examples.example_model_dict, 76
smooth.examples.example_model_emissions, 76
smooth.examples.example_model_external_components, 77
smooth.examples.example_plotting_dicts, 78
smooth.examples.run_optimization_example, 78
smooth.examples.run_smooth_example, 79
smooth.framework, 94
smooth.framework.functions, 90
smooth.framework.functions.calculate_external_costs 81
smooth.framework.functions.debug, 81
smooth.framework.functions.functions, 82
smooth.framework.functions.load_results, 84
smooth.framework.functions.plot_results,

85
smooth.framework.functions.print_results,
86
smooth.framework.functions.save_results,
86
smooth.framework.functions.update_annuities,
86
smooth.framework.functions.update_fitted_cost,
87
smooth.framework.run_smooth, 90
smooth.framework.simulation_parameters,
93
smooth.optimization, 102
smooth.optimization.run_optimization,
95

Index

A

add_to_oemof_model() (*smooth.components.component.Component method*), 7
add_to_oemof_model() (*smooth.components.component_air_source_heat_pump.AirSourceHeatPump method*), 10
add_to_oemof_model() (*smooth.components.component_battery.Battery method*), 13
add_to_oemof_model() (*smooth.components.component_biogas_converter.BiogasConverter method*), 15
add_to_oemof_model() (*smooth.components.component_biogas_smr_psa.BiogasSmrPsa method*), 17
add_to_oemof_model() (*smooth.components.component_compressor_h2.CompressorH2 method*), 20
add_to_oemof_model() (*smooth.components.component_electric_heater.ElectricHeater method*), 21
add_to_oemof_model() (*smooth.components.component_electrolyzer.Electrolyzer method*), 24
add_to_oemof_model() (*smooth.components.component_electrolyzer_waste_heat.ElectrolyzerWasteHeat method*), 29
add_to_oemof_model() (*smooth.components.component_energy_demand_from_csv.EnergyDemandFromCsv method*), 31
add_to_oemof_model() (*smooth.components.component_energy_source_from_csv.EnergySourceFromCsv method*), 32
add_to_oemof_model() (*smooth.components.component_fuel_cell_chp.FuelCellChp method*), 35
add_to_oemof_model() (*smooth.components.component_gas_engine_chp_biogas.GasEngineChpBiogas method*), 39
add_to_oemof_model() (*smooth.components.component_gate.Gate method*), 40
add_to_oemof_model() (*smooth.components.component_h2_chp.H2Chp method*), 43
add_to_oemof_model() (*smooth.components.component_h2_refuel_cooling_system.H2Re method*), 42
add_to_oemof_model() (*smooth.components.component_pem_electrolyzer.PemElectrolyz method*), 46
add_to_oemof_model() (*smooth.components.component_power_converter.PowerConverte method*), 47
add_to_oemof_model() (*smooth.components.component_sink.Sink method*), 48
add_to_oemof_model() (*smooth.components.component_storage_h2.StorageH2 method*), 51
add_to_oemof_model() (*smooth.components.component_stratified_thermal_storage.Strat method*), 54
add_to_oemof_model() (*smooth.components.component_supply.Supply method*), 57
add_to_oemof_model() (*smooth.components.component_trailer_gate.TrailerGate method*), 58
add_to_oemof_model() (*smooth.components.component_trailer_gate_cascade.TrailerGa method*), 59
add_to_oemof_model() (*smooth.components.component_trailer_h2_delivery.TrailerH2De method*), 61
add_to_oemof_model() (*smooth.components.component_trailer_h2_delivery_cascade.Tra method*), 63

add_to_oemof_model() (*smooth.components.component_trailer_h2_delivery_single.TrailerH2DeliverySingle method*), 65
AirSourceHeatPump (*class in smooth.components.component_air_source_heat_pump*), 9
AttributeVariation (*class in smooth.optimization.run_optimization*), 98

B

Battery (*class in smooth.components.component_battery*), 12
BiogasConverter (*class in smooth.components.component_biogas_converter*), 15
BiogasSmrPsa (*class in smooth.components.component_biogas_smr_psa*), 17

C

calc_annual_emissions() (*in module smooth.framework.functions.update_annuities*), 86
calc_annuity() (*in module smooth.framework.functions.update_annuities*), 86
calculate_compressibility_factor() (*in module smooth.components.component_compressor_h2*), 20
calculate_losses() (*smooth.components.component_stratified_thermal_storage.StratifiedThermalStorage method*), 55
calculate_storage_u_value() (*smooth.components.component_stratified_thermal_storage.StratifiedThermalStorage method*), 55
check_flows() (*smooth.components.component_battery.Battery method*), 13
check_validity() (*smooth.components.component.Component method*), 7
check_validity() (*smooth.components.external_component.ExternalComponent method*), 67
choose_valid_dict() (*in module smooth.framework.functions.functions*), 82
Component (*class in smooth.components.component*), 7
CompressorH2 (*class in smooth.components.component_compressor_h2*), 19
compute_fitness() (*smooth.optimization.run_optimization.Optimization method*), 101
conversion_fun_ely() (*smooth.components.component_electrolyzer.Electrolyzer method*), 24
conversion_fun_ely() (*smooth.components.component_electrolyzer_waste_heat.ElectrolyzerWasteHeat method*), 29
conversion_fun_thermal() (*smooth.components.component_electrolyzer_waste_heat.ElectrolyzerWasteHeat method*), 29
costs_for_ext_components() (*in module smooth.framework.functions.calculate_external_costs*), 81
create_component_obj() (*in module smooth.framework.functions.functions*), 82
crossover() (*in module smooth.optimization.run_optimization*), 99
cut_suffix() (*in module smooth.framework.functions.functions*), 82
cut_suffix_loop() (*in module smooth.framework.functions.functions*), 82

E

ElectricHeater (*class in smooth.components.component_electric_heater*), 21
Electrolyzer (*class in smooth.components.component_electrolyzer*), 23
ElectrolyzerWasteHeat (*class in smooth.components.component_electrolyzer_waste_heat*), 28
ely_voltage_u_act() (*smooth.components.component_electrolyzer.Electrolyzer method*), 24
ely_voltage_u_ohm() (*smooth.components.component_electrolyzer.Electrolyzer method*), 24
ely_voltage_u_rev() (*smooth.components.component_electrolyzer.Electrolyzer method*), 25
EnergyDemandFromCsv (*class in smooth.components.component_energy_demand_from_csv*), 31
EnergySourceFromCsv (*class in smooth.components.component_energy_source_from_csv*), 32
err_callback() (*smooth.optimization.run_optimization.Optimization method*), 101
ExternalComponent (*class in smooth.components.external_component*), 66
extract_flow_per_bus() (*in module smooth.framework.functions.functions*), 83

F

fitness (*smooth.optimization.run_optimization.Individual attribute*), 99

fitness_function() (*in module smooth.optimization.run_optimization*), 99
FuelCellChp (*class in smooth.components.component_fuel_cell_chp*), 34

G

GasEngineChpBiogas (*class in smooth.components.component_gas_engine_chp_biogas*), 38
Gate (*class in smooth.components.component_gate*), 40
generate_results() (*smooth.components.component.Component method*), 7
generate_results() (*smooth.components.external_component.ExternalComponent method*), 67
get_cell_temp() (*smooth.components.component_electrolyzer.Electrolyzer method*), 25
get_costs_and_art_costs() (*smooth.components.component.Component method*), 8
get_date_time_index() (*in module smooth.framework.functions.functions*), 83
get_dependant_value() (*in module smooth.framework.functions.update_fitted_cost*), 87
get_df_debug() (*in module smooth.framework.functions.debug*), 81
get_diameter() (*smooth.components.component_stratified_thermal_storage.StratifiedThermalStorage method*), 55
get_el_energy_by_h2() (*smooth.components.component_fuel_cell_chp.FuelCellChp method*), 35
get_electrical_energy_by_bg() (*smooth.components.component_gas_engine_chp_biogas.GasEngineChpBiogas method*), 39
get_electrical_energy_by_h2() (*smooth.components.component_h2_chp.H2Chp method*), 43
get_electricity_by_power() (*smooth.components.component_electrolyzer.Electrolyzer method*), 25
get_exp() (*in module smooth.framework.functions.update_fitted_cost*), 87
get_foreign_state_value() (*smooth.components.component.Component method*), 8
get_free() (*in module smooth.framework.functions.update_fitted_cost*), 88
get_h2_production_by_electricity() (*smooth.components.component_pem_electrolyzer.PemElectrolyzer method*), 46
get_mass() (*smooth.components.component_storage_h2.StorageH2 method*), 52
get_mass_and_temp() (*smooth.components.component_electrolyzer.Electrolyzer method*), 25
get_mass_produced_by_current_state() (*smooth.components.component_electrolyzer.Electrolyzer method*), 25
get_poly() (*in module smooth.framework.functions.update_fitted_cost*), 88
get_pressure() (*smooth.components.component_storage_h2.StorageH2 method*), 52
get_sim_time_span() (*in module smooth.framework.functions.functions*), 83
get_spec() (*in module smooth.framework.functions.update_fitted_cost*), 88
get_th_energy_by_h2() (*smooth.components.component_fuel_cell_chp.FuelCellChp method*), 35
get_thermal_energy_by_bg() (*smooth.components.component_gas_engine_chp_biogas.GasEngineChpBiogas method*), 39
get_thermal_energy_by_h2() (*smooth.components.component_h2_chp.H2Chp method*), 43
get_volume() (*smooth.components.component_storage_h2.StorageH2 method*), 52
get_volume() (*smooth.components.component_stratified_thermal_storage.StratifiedThermalStorage method*), 55
get_waste_heat() (*smooth.components.component_electrolyzer.Electrolyzer method*), 30
get_waste_heat_energy_by_electricity() (*smooth.components.component_pem_electrolyzer.PemElectrolyzer method*), 46
gradient_ascent() (*smooth.optimization.run_optimization.Optimization method*), 101

H

H2Chp (*class in smooth.components.component_h2_chp*), 42
H2Dispenser (*class in smooth.components.external_component_h2_dispenser*), 68
H2RefuelCoolingSystem (*class in smooth.components.component_h2_refuel_cooling_system*), 41
handle_close() (*smooth.optimization.run_optimization.PlottingProcess method*), 100
hover() (*smooth.optimization.run_optimization.PlottingProcess method*), 100

I

Individual (*class in smooth.optimization.run_optimization*), 98
Individual.IndividualIterator (*class in smooth.optimization.run_optimization*), 99
interval_time_index() (*in module smooth.framework.functions.functions*), 83

L

load_results() (*in module smooth.framework.functions.load_results*), 84

M

main() (*smooth.optimization.run_optimization.PlottingProcess method*), 100
mutate() (*in module smooth.optimization.run_optimization*), 99

O

Optimization (*class in smooth.optimization.run_optimization*), 100

P

PemElectrolyzer (*class in smooth.components.component_pem_electrolyzer*), 45
plot_smooth_results() (*in module smooth.framework.functions.plot_results*), 85
PlottingProcess (*class in smooth.optimization.run_optimization*), 100
PowerConverter (*class in smooth.components.component_power_converter*), 47
prepare_simulation() (*smooth.components.component.Component method*), 8
prepare_simulation() (*smooth.components.component_battery.Battery method*), 13
prepare_simulation() (*smooth.components.component_biogas_smr_psa.BiogasSmrPsa method*), 17
prepare_simulation() (*smooth.components.component_compressor_h2.CompressorH2 method*), 20
prepare_simulation() (*smooth.components.component_storage_h2.StorageH2 method*), 52
prepare_simulation() (*smooth.components.component_stratified_thermal_storage.StratifiedThermalStorage method*), 55
prepare_simulation() (*smooth.components.component_supply.Supply method*), 57
prepare_simulation() (*smooth.components.component_trailer_gate.TrailerGate method*), 58
prepare_simulation() (*smooth.components.component_trailer_gate_cascade.TrailerGa method*), 59
prepare_simulation() (*smooth.components.component_trailer_h2_delivery.TrailerH2D method*), 61
prepare_simulation() (*smooth.components.component_trailer_h2_delivery_cascade.Tra method*), 63
prepare_simulation() (*smooth.components.component_trailer_h2_delivery_single.Trail method*), 65
print_smooth_results() (*in module smooth.framework.functions.print_results*), 86

R

read_data_file() (*in module smooth.framework.functions.functions*), 83
replace_at_idx() (*in module smooth.framework.functions.functions*), 84
run() (*smooth.optimization.run_optimization.Optimization method*), 102
run_optimization() (*in module smooth.optimization.run_optimization*), 102
run_smooth() (*in module smooth.framework.run_smooth*), 93

S

save_intermediate_result() (*smooth.optimization.run_optimization.Optimization method*), 101
save_results() (*in module smooth.framework.functions.save_results*), 86
sensible_and_latent_heats() (*smooth.components.component_electrolyzer_waste_heat.Electro method*), 30
set_fitness() (*smooth.optimization.run_optimization.Optimization method*), 101
set_parameters() (*smooth.components.component.Component method*), 8
set_parameters() (*smooth.components.external_component.Externa method*), 67
set_parameters() (*smooth.framework.simulation_parameters.Simula method*), 93
show_debug() (*in module smooth.framework.functions.debug*), 82

SimulationParameters (*class in smooth.framework.simulation_parameters*), 93
Sink (*class in smooth.components.component_sink*), 48
smooth.components (*module*), 68
smooth.components.component (*module*), 7
smooth.components.component_air_source_heat_pump (*module*), 9
smooth.components.component_battery (*module*), 11
smooth.components.component_biogas_converter (*module*), 14
smooth.components.component_biogas_smr_psa (*module*), 15
smooth.components.component_compressor_h2 (*module*), 18
smooth.components.component_electric_heater (*module*), 20
smooth.components.component_electrolyzers (*module*), 22
smooth.components.component_electrolyzers_vaste_heat (*module*), 25
smooth.components.component_energy_demand_from_csv (*module*), 30
smooth.components.component_energy_source_from_csv (*module*), 31
smooth.components.component_fuel_cell_chp (*module*), 32
smooth.components.component_gas_engine_chp_biogas (*module*), 36
smooth.components.component_gate (*module*), 40
smooth.components.component_h2_chp (*module*), 42
smooth.components.component_h2_refuel_container_system (*module*), 40
smooth.components.component_pem_electrolyzer (*module*), 43
smooth.components.component_power_converter (*module*), 47
smooth.components.component_sink (*module*), 48
smooth.components.component_storage_h2 (*module*), 48
smooth.components.component_stratified_thermal_storage (*module*), 52
smooth.components.component_supply (*module*), 56
smooth.components.component_trailer_gate (*module*), 57
smooth.components.component_trailer_gate_mass_flow (*module*), 58
smooth.components.component_trailer_h2_delivery (*module*), 59
smooth.components.component_trailer_h2_delivery_sim (*module*), 61
smooth.components.component_trailer_h2_delivery_sim (*module*), 64
smooth.components.component_var_grid (*module*), 66
smooth.components.external_component (*module*), 66
smooth.components.external_component_h2_dispenser (*module*), 67
smooth.examples (*module*), 79
smooth.examples.example_model (*module*), 69
smooth.examples.example_model_costs (*module*), 70
smooth.examples.example_model_dict (*module*), 76
smooth.examples.example_model_emissions (*module*), 76
smooth.examples.example_model_external_components (*module*), 77
smooth.examples.example_plotting_dicts (*module*), 78
smooth.examples.run_optimization_example (*module*), 78
smooth.examples.run_smooth_example (*module*), 79
smooth.framework (*module*), 94
smooth.framework.functions (*module*), 90
smooth.framework.functions.calculate_external_costs (*module*), 81
smooth.framework.functions.debug (*module*), 81
smooth.framework.functions.functions (*module*), 82
smooth.framework.functions.load_results (*module*), 84
smooth.framework.functions.plot_results (*module*), 85
smooth.framework.functions.print_results (*module*), 86
smooth.framework.functions.save_results (*module*), 86
smooth.framework.functions.update_annuities (*module*), 86
smooth.framework.functions.update_fitted_cost (*module*), 87
smooth.framework.run_smooth (*module*), 90
smooth.framework.simulation_parameters (*module*), 93
smooth.optimization (*module*), 102
smooth.optimization.run_optimization (*module*), 95
solidity_result (*smooth.optimization.run_optimization.Individual attribute*), 99
StorageH2Cascade (*class in*

smooth.components.component_storage_h2), 50
StratifiedThermalStorage (*class in smooth.components.component_stratified_thermal_storage*), 53
Supply (*class in smooth.components.component_supply*), 56

T

TrailerGate (*class in smooth.components.component_trailer_gate*), 57
TrailerGateCascade (*class in smooth.components.component_trailer_gate_cascade*), 58
TrailerH2Delivery (*class in smooth.components.component_trailer_h2_delivery*), 60
TrailerH2DeliveryCascade (*class in smooth.components.component_trailer_h2_delivery_cascade*), 63
TrailerH2DeliverySingle (*class in smooth.components.component_trailer_h2_delivery_single*), 65

U

update_annuities() (*in module smooth.framework.functions.update_annuities*), 87
update_constraints() (*smooth.components.component.Component method*), 8
update_constraints() (*smooth.components.component_electrolyzer_waste_heat.ElectrolyzerWasteHeat method*), 30
update_constraints() (*smooth.components.component_fuel_cell_chp.FuelCellChp method*), 35
update_constraints() (*smooth.components.component_gas_engine_chp_biogas.GasEngineChpBiogas method*), 39
update_constraints() (*smooth.components.component_h2_chp.H2Chp method*), 43
update_constraints() (*smooth.components.component_pem_electrolyzer.PemElectrolyzer method*), 46
update_cost() (*in module smooth.framework.functions.update_fitted_cost*), 89
update_emissions() (*in module smooth.framework.functions.update_fitted_cost*), 89
update_external_annuities() (*in module smooth.framework.functions.update_annuities*), 87
update_financials() (*in module smooth.framework.functions.update_fitted_cost*), 89
update_flows() (*smooth.components.component.Component method*), 8
update_flows() (*smooth.components.component_battery.Battery method*), 13
update_flows() (*smooth.components.component_electrolyzer_waste_h method*), 30
update_flows() (*smooth.components.component_fuel_cell_chp.FuelC method*), 36
update_flows() (*smooth.components.component_gas_engine_chp_bio method*), 39
update_flows() (*smooth.components.component_h2_chp.H2Chp method*), 43
update_flows() (*smooth.components.component_pem_electrolyzer.Pe method*), 46
update_nonlinear_behaviour() (*smooth.components.component_electrolyzer.Electrolyzer method*), 25
update_nonlinear_behaviour() (*smooth.components.component_electrolyzer_waste_heat.Electro method*), 30
update_states() (*smooth.components.component.Component method*), 9
update_states() (*smooth.components.component_battery.Battery method*), 14
update_states() (*smooth.components.component_compressor_h2.Co method*), 20
update_states() (*smooth.components.component_electrolyzer.Electro method*), 25
update_states() (*smooth.components.component_storage_h2.Storage method*), 52
update_states() (*smooth.components.component_stratified_thermal_ method*), 56
update_var_costs() (*smooth.components.component.Component method*), 9
update_var_costs() (*smooth.components.component_trailer_gate.TrailerGate method*), 58
update_var_emissions() (*smooth.components.component.Component method*), 9

V

values (*smooth.optimization.run_optimization.Individual attribute*), 99
VarGrid (*class in smooth.components.component_var_grid*), 66