Documentation of the Simulation Framework OpEnCellS

Release 0.1
Berlin, den November 20, 2020

Fabian Schmid

Technical University of Berlin
Institute for Energy Engineering
Chair for Energy Process Engineering and Conversion Technologies for Renewable Energies

fabian.schmid@tu-berlin.de

Contents

Chapter 1

OpEnCellS - Open energy cell simulation

1.1 Introduction

OpEnCellS (Open energy cell simulation) is a open-source modeling framework for the simulation of energy systems. The tool is developed by Fabian Schmid, written in Python and follows an object-oriented modeling approach.

OpEnCellS enables a detailed simulation of various renewable energy technologies with a temporal resolution of one minute or one hour. Electricity sector components involves models for photovoltaic, power components (Pulse-Width-Modulation (PWM) and Maximum Power Point Tracker (MPPT) charge controllers, Battery Management Systems (BMS)) and batteries (lead acid and lithium-ion). Component models of the heat sector involve solarthermal collectors, pipes and stratified heat storages are currently under development. Future releases will include hydrogen and wind turbine systems.

All components include a degradation or aging models to determine component lifetime and possible degradation in the case of photovoltaic, battery or solarthermal component.

A generic energy system can be constructed from the provided component classes. Technical and economical key performance indicators can be derived and enables the evaluation and assessment of the simulated energy system. Furthermore it can be used as a base for a system optimization (e.g. multi-objective optimization with NSGA-II or NSGA-III).

1.2 Quickstart

It is recommended to use Python Anaconda distribution (check this Link) and use OpEnCellS in a virtual environment (can be generated with conda or the Anaconda Navigator). For more information of managing virtual environments see this Link. But this is optional.

You can simply clone or download OpEnCellS from github. Additional Python libaries you need to install manually are the following:

- pvlib libary version 0.7.1, see this Link

OpEnCellS can be simply started with executing the *main.py* under the folder *projects/sample*. This script starts a sample simulation with some basic evaluations of a sample energy system configuration. System configuration is defined in the file *simulation.py*. All input parameters (components parameter, load and environmental timeseries profiles) are loaded from the folder *input*.

1.3 Basic structure of SimCell

OpEnCellS follows a object-oriented programming style, which allows a generic combination of different system components. It is a time continuous simulation, means that for each simualation timestep each model is calculated and the component status is defined.

The basic structure of the programm can be seen in the following figure:

INSERT FIGURE - class diagramm of simulation with all components included?

1.4 Contact

For any comments or questions please contact: fabian.schmid@tu-berlin.de

Chapter 2

Code structure

The following sections give an overview and explanation on the central Simulation class and related methods and classes (compare 2.1 and the main file to initialize a simulation instance (compare 2.2. For a detailed description of the component classes it is referred to the API documentation.

2.1 Simulation class

The central simulation class in the file *simulation.py* defines the system components and its connections. The basic structure is presented with the sample simulation. First basic simulation parameters needs to be defined, second the environmental and load classes are stated and third the component classes are defined with its input links. The input link defines the class, which provides the input power to the component. In case of the photovoltaic class the input link is the environmental class, which determines the solar irradiation. Finally all classes needs to be added to the Simulatable class in order to make them simulatable.

Listing 2.1 Central Simulation class for energy system definition.

```python
import pvlib
from datetime import datetime

from simulatable import Simulatable
from environment import Environment
from components.load import Load
from components.photovoltaic import Photovoltaic
from components.power_component import Power_Component
from components.power_junction import Power_Junction
from components.battery import Battery

class Simulation(Simulatable):

    def __init__(self,
                 simulation_steps,
                 timestep):

        # [Wp] Installed PV power
        self.pv_peak_power = 120
        # [Wh] Installed battery capacity
        self.battery_capacity = 600
        #  PV orientation : tuble of floats. PV oriantation with:
        self.pv_orientation = (0,0)
        # System location
        self.system_location =
        pvlib.location.Location(latitude=-3.386925,
                                longitude=36.682995,
                                tz='Africa/Dar_es_Salaam',
                                altitude=1400)
        # Number of simulation timesteps
        self.simulation_steps = simulation_steps
        # [s] Simulation timestep
        self.timestep = timestep


```

```
        ## Initialize classes
        # Environment class
        self.env =
        Environment(timestep=self.timestep,
                    system_orientation=self.pv_orientation,
                    system_location=self.system_location)

        # load class
        self.load =
        Load()

        # Component classes
        self.pv =
        Photovoltaic(timestep=self.timestep,
                     peak_power=self.pv_peak_power,
                     controller_type='mppt',
                     env=self.env,
                     file_path='data/components/photovoltaic_resonix_120Wp.json')

        self.charger =
        Power_Component(timestep=self.timestep,
                        power_nominal=self.pv_peak_power,
                        input_link=self.pv,
                        file_path='data/components/power_component_mppt.json')

        self.power_junction =
        Power_Junction(input_link_1=self.charger,
                       input_link_2=None,
                       load=self.load)

        self.battery_management =
        Power_Component(timestep=self.timestep,
                        power_nominal=self.pv_peak_power,
                        input_link=self.power_junction,
                        file_path='data/components/power_component_bms.json')

        self.battery =
        Battery(timestep=self.timestep,
                capacity_nominal_wh=self.battery_capacity,
                input_link=self.battery_management,
                env=self.env,
                file_path='data/components/battery_lfp.json')

        ## Initialize Simulatable class and define needs_update initially to True
        self.needs_update = True

        Simulatable.__init__(self, self.env, self.load, self.pv, self.charger,
                             self.power_junction, self.battery_management, self.battery)
```

The simulatable class in the file *simulatable.py* enables the call of each component class for every timestep with its methods *start* (starting of simulation and set start values), *end* (end of simulation) and *update* (simulation goes one step forward and re-calculates each component class). Childs* stands for all component classes, which shall be made simulatable, compare Listings 2.2 line 3. *Self.time* is the timestep of the simulation, which increase by one for every step.

Listing 2.2 Simulatable class to make simulation simulatable.

```
1  class Simulatable:
2
3      def __init__(self, *childs):
4          self.time = -1
5          self.childs = list(childs)
6
7      def calculate(self):
8          pass
9
10
11     def start(self):
12         # Set time index to zero
13         self.time = 0
14         # Calls start method for all simulatable childs
15         for child in self.childs:
16             if isinstance(child, Simulatable):
17                 child.start()
18
19     def end(self):
20         # Set time index back to 0
21         self.time = 0
22         # Calls end method for all simulatable childs
23         for child in self.childs:
24             if isinstance(child, Simulatable):
25                 child.end()
26
27     def update(self):
28         # Call null method
29         self.calculate()
30
31         # Update time parameters with +1
32         self.time += 1
```

```
        # Calls update method for all simulatable childs
        for child in self.childs:
            if isinstance(child, Simulatable):
                child.update()
```

The simulation class in *simulation.py* has one method, which is *simulate*. This runs the actual simulation step by step, using the methods start(), update() and end() of the simulatable class, which owns all component classes as childs. Initially the variable *self.needs update* is set to true (compare listing 2.1 line 80 and will be only set to false at the end of the simulation. The code is displayed in the following listing:

Listing 2.3 Central method of simulation class to run actual simulation step by step.

```
    def simulate(self):

        ## Initialization of list containers to store simulation results
        # Timeindex
        self.timeindex = list()
        # Load demand
        self.load_power_demand = list()
        # PV
        self.pv_power = list()
        self.pv_temperature = list()
        # charger
        self.charger_power = list()
        self.charger_efficiency = list()
        # Power junction
        self.power_junction_power = list()
        # BMS
        self.battery_management_power = list()
        self.battery_management_efficiency = list()
        # Battery
        self.battery_power = list()
        self.battery_efficiency = list()

        ## As long as needs_update = True simulation takes place
        if self.needs_update:

            ## Timeindex from irradiation data file
            time_index = self.env.time_index
            ## pvlib: irradiation and weather data
            self.env.load_data()
            ## pvlib: pv power
            self.pv.load_data()

            ## simlation start: needs_update is true
            self.start()

            ## Iteration over all simulation steps
            for t in range(0, self.simulation_steps):
                ## Call components calculate method and go one simulation step further
                self.update()

                ## Store simulation results in list containers
                # Time index
                self.timeindex.append(time_index[t])
                # Load demand
                self.load_power_demand.append(self.load.power)
                # PV
                self.pv_power.append(self.pv.power)
                self.pv_temperature.append(self.pv.temperature)
                # charger
                self.charger_power.append(self.charger.power)
                self.charger_efficiency.append(self.charger.efficiency)
                # Power jundtion
                self.power_junction_power.append(self.power_junction.power)
                # BMS
                self.battery_management_power.append(self.battery_management.power)
                self.battery_management_efficiency.append(self.battery_management.efficiency)
                # Battery
                self.battery_power.append(self.battery.power_battery)
                self.battery_efficiency.append(self.battery.efficiency)

            ## Simulation over: set needs_update to false and call end method
            self.needs_update = False
            self.end()
```

2.2 Main file

The *Main.py* file will initialize an instance of the simulation class, load irradiaion and load data with the defined classes load and env, compare listing 2.1 line 38/39 and 44/45, which will use the helper class data loader for this purpose (compare **??**). Finally the simulate method of the defined simulation instance, compare listing 2.4 line 31 is called.

Listing 2.4 Main file to define the simulation instance.

```
1  import pandas as pd
2  from collections import OrderedDict
3  from simulation import Simulation
4
5  # Simulation timestep in seconds
6  timestep = 60*60
7  # Simulation number of timestep
8  simulation_steps = 24*365
9
10 ## Create Simulation instance
11 sim = Simulation(simulation_steps=simulation_steps,
12                  timestep=timestep)
13
14 ## load environmental data
15 #Load timeseries irradiation data
16 sim.env.meteo_irradiation.read_csv(
17  file_name='data/env/SoDa_Cams_Radiation_h_2006-2016_Arusha.csv',
18     start=0,
19     end=simulation_steps)
20 #Load weather data
21 sim.env.meteo_weather.read_csv(
22  file_name='data/env/SoDa_MERRA2_Weather_h_2006-2016_Arusha.csv',
23     start=0,
24     end=simulation_steps)
25 #Load load demand data
26 sim.load.load_demand.read_csv(file_name='data/load/load_dummy_h.csv',
27                               start=0,
28                               end=24)
29
30 ## Call Main Simulation method
31 sim.simulate()
32
33
34 ## Simulation resultd
35 results_power = pd.DataFrame(
36     data=OrderedDict({'pv_power':sim.pv_power,
37                       'charger_power':sim.charger_power,
38                       'load_power':sim.load_power_demand,
39                       'power_junction_power':sim.power_junction_power,
40                       'battery_management_power':sim.battery_management_power,
41                       'battery_power':sim.battery_power,
42                       'battery_soc':sim.battery_state_of_charge}), index=sim.timeindex)
```

Chapter 3

Mathematical models - Electric components

3.1 Photovoltaic

The photovoltaic model is based on the pvlib libary version 0.7.1 available under [1]. The model can be structured in following submodels:

- Photovoltaic thermal model
- Photovoltaic power model
- Photovoltaic degradation model

3.1.1 Photovoltaic thermal model

The photovoltaic thermal model is based on the Sandia array Performance Model and is given by following pair of equations [**?**]:

$$T_m = G \cdot e^{a+b\cdot v_{\text{wind}}} + T_{\text{a}} \tag{3.1}$$

$$T_{\text{c}} = T_{\text{m}} + \frac{G}{G_{\text{STC}}} \cdot \Delta T \tag{3.2}$$

Inputs to the model are plane-of-array irradiance G in $W \cdot m^{-2}$ and ambient air temperature T_a in K. Ambient temperature in provided by environment class in Kelvin. Model parameters depend both on the module construction and its mounting. The model returns the photvoltaic cell temperature in C.

For more Details on using the $pvlib.temperature.sapm - cell$ method it is referred to[2].

[1] https://pvlib-python.readthedocs.io/en/stable/index.html
[2] https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.temperature.sapm_cell.html

3.1.2 Photovoltaic power model

The photvoltaic power model consists of:

- Photovoltaic power model with MPPT assumption
- Photovoltaic power model with MPPT assumption

Photovoltaic power model with MPPT assumption The simple and fast Photovoltaic power model with MPPT assumption is based on the PVWatts Version 5 model [?]. It is defined by following equation:

$$P_{\text{module, MPP}} = \frac{G_{\text{poa, effective}}}{G_{\text{STC}}} \cdot (P_{\text{dco}} \cdot (1 + \gamma_{\text{pdc0}} \cdot (T_{\text{cell}} - T_{\text{STC}})) \tag{3.3}$$

Photovoltaic cell temperature needs to be specified in C. Note that the P_{dc0} is also used as a symbol in $pvwatts_a c()$. P_{dc0} in this function refers to the DC power of the modules at reference conditions. P_{dc0} in $pvwatts_a c()$ refers to the DC power input limit of the inverter. For more Details on using the $pvlib.pvsystem.pvwatts - dc$ method it is referred to [3].

Photovoltaic power model with PWM assumption The more detailed Photovoltaic power model with PWM assumption computes all parameters for the single diode model and the construction of the VI curve based on the method of De Soto et al. [?]. The five values returned by the method are used to further calculate the VI curve and the photovoltaic power at a specfic voltage. Photovoltaic cell temperature needs to be specified in C.

$$I = I_{\text{L}} - I_{\text{D}} - I_{\text{sh}} \tag{3.4}$$

$$I = I_{\text{L}} - I_0 \cdot \left(e^{\frac{(V + I \cdot R_{\text{s}})}{a}} - 1\right) - \frac{V + I \cdot R_{\text{s}}}{R_{\text{sh}}} \tag{3.5}$$

$$P_{\text{module, PWM}} = I \cdot V \tag{3.6}$$

a: ideality factor, I_{L}: light current, I_0: diode reverse saturation current, R_{s}: series resistance, R_{sh}: shunt resistance.

3.1.3 Photovoltaic degradation model

The photovoltaic degradation model assumes a annual power degradation, default value is $0.5\,\% \cdot a^{-1}$. Timestep is the temporal resolution of the simulation in seconds.

$$P_{\text{module, nominal}}(t) = (1 - \frac{0.005}{8760 \cdot \frac{3600}{\text{timestep}}}) \cdot P_{\text{module, nominal}}(t - 1) \tag{3.7}$$

Battery temperature is determined and ambient temperature input values are given in K.

[3] https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.pvsystem.pvwatts_dc.html

3.2 Battery

The battery model can be structured in following submodels:

- Battery thermal model
- Batter stationary model
- Battery degradation model

3.2.1 Battery thermal model

The battery thermal model is based on the heat equation by Bernardi et al., but only heat flows due to ohmic losses and heat transport to the environment are considered.

$$\dot{Q} = \dot{Q}_{\text{irr}} + \dot{Q}_{\text{rev}} + \dot{Q}_{\text{reakt}} + \dot{Q}_{\text{mix}} \tag{3.8}$$

$$\frac{dT}{dt} = \frac{(h \cdot A \cdot (T_{\text{a}} - T_{\text{B}}) + P_{\text{B,loss}}) \cdot \Delta t}{m \cdot c_p} \tag{3.9}$$

$$T_{\text{B}}(t+1) = \frac{h \cdot A \cdot (T_{\text{a}}(t) - T_{\text{B}}(t)) + P_{\text{B,loss}}}{m \cdot c_p} + T_{\text{B}}(t) \tag{3.10}$$

3.2.2 Battery stationary model

The battery stationary model provides charge and discharge efficiency, its charge and discharge boundaries and the determination of the State of Charge. It includes as well the charge and discharge algorithms, which decides if battery is capable of supplying load demand / charge power.

The battery efficiency can be defined as a power dependent equation, based on the following equation:

$$\begin{aligned} \eta_{\text{CH}} &= a \cdot \text{C-Rate} + b \\ \eta_{\text{DCH}} &= c \cdot \text{C-Rate} + d \end{aligned} \tag{3.11}$$

The charge and discharge boundaries can be defined as a power dependent boundary, based on the following equation:

$$\begin{aligned} SoC_{\text{CH, cut off}} &= e \cdot \text{C-Rate} + f \\ SoC_{\text{DCH, cut off}} &= g \cdot \text{C-Rate} + h \end{aligned} \tag{3.12}$$

The coefficients a to f need to be parameterized externally. This can be done by charge and discharge curves for at least three different C-Rates, which is normally provided in most battery datasheets.

The battery State of Charge is defined by follwing equation:

$$SoC = SoC_{t-1} + \frac{(P - P_{\text{loss}} - P_{\text{self-discharge}}) \cdot \Delta t}{C_{\text{current}}} \tag{3.13}$$

The equation takes charge and discharge losses, self-discharge effect and the current battery capacity (with degradation) into account.

The charge and discharge algorithm compares the boundaries and the current and future State of Charge and decides weather a charge or discharge can be set to the battery.

Data: Input component and its power flow
Result: Battery power and state of charge
initialization;
Calculate *battery power* (efficiency, power loss);
Calculate *battery ch-dch boundary* ;
Calculate *battery state of charge*;
if *discharge case* **then**
    if *state of charge* $<$ *ch-dch boundary* **then**
        re-calculate *battery power* (extractable till boundary);
        if *battery yet fully discharged* **then**
            state of charge = state of charge OLD
        else
            state of charge = ch-dch boundary
        end
        re-calculate *input link power* and its efficiency;
    end
end
if *charge case* **then**
    if *state of charge* $>$ *ch-dch boundary* **then**
        re-calculate *battery power* (chargeable till boundary);
        if *battery yet fully charged* **then**
            state of charge = state of charge OLD
        else
            state of charge = ch-dch boundary
        end
        re-calculate *input link power* and its efficiency;
    end
end

Algorithm 1: Battery charge discharge algorithm

Because of the current battery charge discharge algorithm the battery class is combined with the input link class, which is assumed to be a Battery Management System (BMS). In case of battery technologies, which need no BMS, the algorithm needs to be adapted or a generic BMS input class needs to be defined.

3.2.3 Battery degradation model

The battery degradation model can be structured in a calendaric and a cycling aging model. Both models use datasheet data as basis.

The cycling aging model is based on the work by Narayan et al. [?]. It uses the battery cycle-life characteristics as a function of DOD and temperature. It further assumes that for cycling at a given DOD level the cycle life has a linear temperature dependency. The linearity on temperature dependency is used to create polynomial approximation functions.

$$\text{cycle life}(DoD) = p_4 \cdot DoD^4 + p_3 \cdot DoD^3 + p_2 \cdot DoD^2 + p_1 \cdot DoD + p_0 \tag{3.14}$$

$$\text{factor (T)} = p_{l,1} \cdot T + p_{l,2} \tag{3.15}$$

To dynamically integrate the capacity degradation the zero-crossing micro cycle approach is used. A zero-crossing micro cycle is defined as the time between two subsequent status where battery power is zero. After every micro cycle the mean temperature and DoD is determined and the capacity degradation of this micro cycle calculated with the given linear and polynomial equations.

$$\overline{DoD} = \frac{\sum_{n=0}^{N} DoD_i}{N} \tag{3.16}$$

$$\overline{T} = \frac{\sum_{n=0}^{N} T_i}{N} \tag{3.17}$$

$$\text{cycle life} = \text{cycle life}(\overline{DoD}) \cdot \text{factor}(\overline{T}) \tag{3.18}$$

The capacity degradation is then determined using the following equation. First the relative degradation is calculated with the real energy throughput of the micro cycle. Second the real capacity degradation is determined. Timestep is the temporal resolution of the simulation in seconds.

$$\text{cycle life}_{microcycle} = \frac{E_{\text{micro cycle}}}{2 \cdot C_{nominal} \cdot \overline{DoD}} \tag{3.19}$$

$$\text{relative degradation} = \frac{\text{cycle life}_{microcycle}}{\text{cycle life}} \tag{3.20}$$

$$\text{absolute degradation} = \text{relative degradation} \cdot (C_{nominal} - C_{\text{end of life}}) \cdot \frac{timestep}{3600} \tag{3.21}$$

3.3 Power Components

The power component model can be structured in following submodels:

- Power component power model

- Power component degradation model

The model is used for the Pulse-Width Modulation and Maximum Power Point Tracker charge controllers and the Battery Management System.

3.3.1 Power Component power model

The power model determines the power dependent efficeincy of the component according to the model develoed by Schmid and Sauer [?].

$$p_{\text{loss}} = p_{\text{self}} + v_{\text{loss}} \cdot p_{\text{out}} + r_{\text{loss}} \cdot p_{\text{out}}^2 \tag{3.22}$$

$$\eta = \frac{P_{\text{out}}}{P_{\text{in}}} = \frac{P_{\text{out}}}{P_{\text{out}} + P_{\text{loss}}} = \frac{P_{\text{out}}}{P_{\text{out}} + (p_{\text{self}} + v_{\text{loss}} \cdot p_{\text{out}} + r_{\text{loss}} \cdot p_{\text{out}}^2)} \tag{3.23}$$

The parameters $\mathrm{p_{self}}$, $\mathrm{v_{loss}}$ and $\mathrm{r_{loss}}$ are determined with efficeincy curves at different fraction of the nominal power dependent on the input power $\mathrm{P_{in}}$ and output power $\mathrm{P_{out}}$. Following parameters are set for current implemented components.

The efficiency dependent on the output power is defined by:

$$\eta = \frac{P_{\text{out}}}{P_{\text{out}} + (p_{\text{self}} + v_{\text{loss}} \cdot p_{\text{out}} + r_{\text{loss}} \cdot p_{\text{out}}^2)} \tag{3.24}$$

The efficiency dependent on the input power is defined by:

$$\eta^* = \frac{1 + v_{\text{loss}}^*}{2 \cdot r_{\text{loss}}^* \cdot p_{\text{in}}} + \sqrt{\frac{(1 + v_{\text{loss}})^2}{(2 \cdot r_{\text{loss}}^* \cdot p_{\text{in}})^2} + \frac{p_{\text{in}} - p_{\text{self}}^*}{r_{\text{loss}}^* \cdot p_{\text{in}}^2}} \tag{3.25}$$

Parameters marked with * are defined by following equations:

$$\frac{p_{\text{self}}^*}{p_{\text{self}}} = \eta_{\text{N}} \tag{3.26}$$

$$\frac{r_{\text{loss}}^*}{r_{\text{loss}}} = \frac{1}{\eta_{\text{N}}} \tag{3.27}$$

$$\frac{v_{\text{loss}}^*}{v_{\text{loss}}} = 1 \tag{3.28}$$

3.3.2 Power Component degradation model

The power component degradation model assues a constant lifetime of the component without any dynamic degradation. The current default values are 10 years for all power components.

Chapter 4

Mathematical models - Heat components

4.1 Solarthermal

The solarthermal model can be used to conduct energy yield simulations but also detailed temperature development simulations based on differential equations for solar vacuum tube collectors. The model can be structured in following submodels:

- Solarthermal static model
- Solarthermal integral model
- Solarthermal pump control

4.1.1 Solarthermal static model

The static solarthermal model is based on a simple energy balance around the solar collector and is based on the following equation.

$$\dot{Q}_{Coll} = \tau \cdot E \cdot A_{Coll} - \dot{Q}_{Convective} - \dot{Q}_{Radiation} - \dot{Q}_{Reflection} \tag{4.1}$$

With a systematic derivation the collector power can be described by following equation:

$$\dot{Q}_{Coll} = A_{Coll} \cdot \Big(\eta_0(\theta) \cdot G_{poa,effective} - k_0 \cdot (T_{Coll,mean} - T_a) + k_1 \cdot (T_{Coll,mean} - T_a)^2\Big) \tag{4.2}$$

with the optical efficiency of the solar collector η_0, the coefficients k_0 and k_1 and the mean solar collector temperature $T_{Coll,mean}$. Further details can be found in [Quaschning.2015]. All parameters can be found in test reports (e.g. [1]). Further the mean collector needs to be defined, which can be calculated or set to a static value, which is a common approach for simple energy yield simulations.

The optical efficiency is further dependent on the solar angle of incidence $\eta_0(\theta)$. This dependency can be calculated with the following equation:

[1] http://www.solarkeymark.nl/DBF/

$$\eta_0(\theta) = \frac{\eta_0 \cdot K_{dir}(\theta) \cdot G_{poa,dir} + \eta_0 \cdot K_{diff}(\theta) \cdot G_{poa,diff}}{G_{poa,dir} + G_{poa,diff}} \tag{4.3}$$

The correction factors K_{diff} is assumed to be a fixed value, while K_{dir} is dependent on the angle of incidence and separated in a longitudinal and transversal part [Theunissen.1985]:

$$K_{dir}(\theta) = K_{dir,long}(\theta_{long}) \cdot K_{dir,trans}(\theta_{trans}) \tag{4.4}$$

Again all parameters can be obtained from collector test reports, where values for the optical efficiency η_0, K_{diff} and values for $K_{dir,long}(\theta_{long})$ and $K_{dir,trans}(\theta_{trans})$ for specific angel of incidence are given. In case another than the given angle needs to be simulated, a linear interpolation is assumed inside the model.

The obtained collector power is a theoretical power, which is not automatically the power which can bus supplied to the system/heat storage. Because it can be restricted due to full heat storage or too low collector flow temperature compared to the heat storage temperature.

4.1.2 Solarthermal integral model

The solarthermal integrale model can be used to model the temperature distribution inside the collector in order to determine the collector output and mean temperature. It is builds on following equation, which is based on an instationary, integral energy balance around the solar collector:

$$\frac{dQ}{dt} = \dot{Q}_{Solar} + \dot{H}_{in} - \dot{H}_{out} - \dot{Q}_{loss} \tag{4.5}$$

with $\dot{Q}_{Solar}$ the incoming energy through the solar irradiation and $\dot{Q}_{loss}$ the energy loss of the collector. Next the following differential equation can be defined:

$$\begin{aligned} \frac{dT_{Coll,mean}}{dt} &= \frac{1}{c_{effective}} \cdot \left(\eta_0(\theta) \cdot G_{poa,effective} - \left(k_0 \cdot (T_{Coll,mean} - T_a) + k_1 \cdot (T_{Coll,mean} - T_a)^2\right)\right) \\ &+ \frac{1}{c_{effective} \cdot A_{Coll}} \cdot \left(\dot{V} \cdot \rho_{fluid} \cdot c_{fluid} \cdot \left(T_{in} - (2 \cdot T_{Coll,mean} - T_{in})\right)\right) \end{aligned} \tag{4.6}$$

with $c_{effective}$ the collector effective heat capacity and the term $T_{out} = 2 \cdot T_{Coll,mean} - T_{in}$.

It is recommended to combine the integral solarthermal model with an integral pipe model to buffer collector output temperature, which increases model accuracy.

4.1.3 Solarthermal pump control

Three different solarthermal pump control models are implemented:

- No control
- Two Point Control
- Matched Flow Control

No Control

Within the *No Control* solarthermal control mechanism no pump control is implemented. The theoretical solarthermal power with equation 4.2. Collector input, mean and output temperature are static and need to be defined externally.

Two Point Control

The *Two Point Control* regulates the solar pump according to the collector output tempertaure and the current heat storage temperature. The collector output temperature needs to be higher than the heat storage to tunrn the pump on. It follows a standard hysteresis curve.

Matched Flow Control

The *Matched Flow Control* regulates the solar pump according to achieve a specified collector target temperature. This is the most common technology. It is modeled with a simple PI controller.

4.2 Pipe

The pipe model consists of an integrale model, which can be used to simulate the temperature development inside a pipe with input and output flow. It is based on following equation:

$$\frac{dQ}{dt} = \dot{H}_{in} - \dot{H}_{out} - \dot{Q}_{loss} \tag{4.7}$$

The following differential equation can be derived from it:

$$\begin{aligned}\frac{dT_{Pipe,out}}{dt} = & \frac{1}{m_{pipe} \cdot c_{pipe} + m_{fluid} \cdot c_{fluid}} \cdot \Big(\dot{V} \cdot \rho_{fluid} \cdot c_{fluid} \cdot (T_{in} - T_{out}) \\ & - k \cdot \pi \cdot d_{pipe,outer} \cdot L \cdot (T_{out} - T_a)\Big)\end{aligned} \tag{4.8}$$

It needs to be mentioned that with this integrale model no delay time can be represented and the output temperature changes directly with input temperature changes.

4.3 Heat storage

The heat storage model can be used to model the temperature distribution and development inside the storage. Following storage model are implemented:

- Perfectly mixed heat storage
- Stratified storage

4.3.1 Perfectly mixed storage

The perfectly mixed heat storage model is based on following equation:

$$
\begin{aligned}
m_{fluid} \cdot c_{fluid} \cdot \frac{dT_{storage}}{dt} &= \dot{H}_{gen1,in} + \dot{H}_{gen2,in} - \dot{H}_{gen1,out} - \dot{H}_{gen2,out} \\
&\quad - \dot{H}_{heating} - \dot{H}_{hotwater} - \dot{Q}_{loss}
\end{aligned}
\tag{4.9}
$$

With the assumption that the generation component input temperatures equals the storage temperature, following equation can be derived:

$$
\begin{aligned}
\frac{dT_{Storage}}{dt} &= \frac{1}{\rho_{fluid} \cdot V_{Storage} \cdot c_{fluid}} \cdot \Big(\dot{V}_{gen1} \cdot \rho_{fluid} \cdot c_{fluid} \cdot (T_{gen1,out} - T_{Storage}) \\
&\quad + \dot{V}_{gen2} \cdot \rho_{fluid} \cdot c_{fluid} \cdot (T_{gen2,out} - T_{Storage}) - \dot{Q}_{heating} - \dot{Q}_{hotwater} \\
&\quad - A \cdot k \cdot (T_{Storage,mean} - T_a)\Big)
\end{aligned}
\tag{4.10}
$$

This is a simple heat storage model, which has multiple drawbacks.

- Flow temperature of generation components as solarthermal collectors is overestimated.
- Heat reservoir of the storage is overestimated.

Therefore it is recommended to use this model in combination with a static solarthermal input temperature. This assumes further that the perfectly mixed storage has a low temperature reservoir, which supplies this static low input temperature to the solarthermal collector adn that the storage has always a good temperature distribution apparent.

4.3.2 Stratified storage

The stratified storage model displays the temperature distribution inside the storage in a more detailed manner. The storage volume is separated into N-finite sub-volumes with a fixed temperature.

The following differential equation describes the system:

For $i = 1 to i = N - 1$:

$$
\begin{aligned}
m_i \cdot \cdot \frac{dT_i}{dt} &= \frac{k \cdot A_i}{c_i} \cdot (T_a - Ti) + \sum_j F_{j,i} \cdot \dot{m}_{Gen,j} \cdot (T_{Gen,j} - T_i) + \sum_j F_{j,i} \cdot \dot{m}_{Con,j} \cdot (T_{Con,j} - T_i) \\
&\quad + \sum_j FT_{j,i} \cdot \dot{m}_{Gen,j} \cdot (T_{i+1} - T_i) + \sum_j FT_{j,i} \cdot \dot{m}_{Con,j} \cdot (T_i - T_{i-1})
\end{aligned}
\tag{4.11}
$$

with Gen representing the generation components and Con the consumption components with the control function F and the transfer function FT.