aristopy documentation

Release 0.9.dev5

Stefan Bruche

Sep 07, 2021

CONTENTS:

The Python package *aristopy* is a framework for modeling and optimizing the design and operation of energy systems. The name of the framework is derived from the great Greek thinker Aristotle. For Aristotle, planning and the wise use of human goods represented great virtues. Transferred to today's time and the design of energy systems, this implies using appropriate tools that support the planning process and contribute to an optimal use of the available resources (money, fuel, etc.).

Selected highlights

- Flexible modeling of energy systems with only a small number of basic components (Source, Sink, Conversion, Bus, Storage) and a comprehensive API.
- Manual scripting of component constraints to enable all types of mathematical modeling classes (linear [LP], mixed-integer linear [MILP], mixed-integer non-linear [MINLP], etc.).
- Declaration of persistent models to quickly run models iteratively after applying small changes (e.g., add an integer-cut constraint).
- Auto-generated visualization of the optimization results with flexible plotting routines.

How to cite aristopy

You are welcome to test *aristopy* and use it for your own purposes. If you publish results based on the application of the package, we kindly ask you to cite this documentation.

The tsam package (time series aggregation module) is used for the clustering of time series.

The package pvlib is applied for calculating solar feed-in time series. Pvlib is not a part of the standard installation of *aristopy* and needs to be obtained separately.

CHAPTER

ONE

INSTALLATION

Before you can create your first optimization model with *aristopy* you need to install the following three essential components:

1. Python and IDE
2. Aristopy package
3. Solver

Python and IDE

The first requirement for using *aristopy* is a working installation of Python on your machine. *Aristopy* is currently tested with Python 3.6 and 3.7. Please use one of the many good tutorials on the internet if you need help with the Python installation. Make sure to add the path of the installation to your system's environment variables to simplify the call of python, pip, etc. from the command line. To enhance your workflow with Python, we also recommend installing an integrated development environment (IDE). PyCharm has served well for us, but there are numerous other useful and free software tools available.

Aristopy package

You can easily install *aristopy* in your current environment via pip by using the following command:

```
>> pip install aristopy
```

Alternatively, you can create a clone of aristopy's GitHub repository (provided git is installed) in a local directory of your machine

```
>> git clone https://git.tu-berlin.de/etus/public/aristopy.git
```

or download a zipped version directly from the GitLab page.

After that, you need to go to your local directory and install *aristopy* by running the setup-file with python

```
>> python setup.py install
```

or using pip install[1].

```
>> pip install -e .[dev]
```

Solver

Note: The installation of *aristopy* does not include any solvers. They need to be obtained separately, in accordance with the properties of your model, the availability of licenses, and your specific preferences.

[1] Argument -e is optional for editable / development mode. Add [dev] if you also want to install the extra-dependencies, i.e. sphinx, pytest, etc.

The availability of a mathematical solver is essential to generate results for your optimization problem. You need to ensure the solver of choice is suitable for your model's mathematical class (e.g., if you added non-linear correlations, you need to use a solver for non-linear problems). We refer to the common literature for further information on mathematical modeling in general.

There is a great variety of solvers available on the market. For the use with *aristopy* you have to consider that the solver interface must be usable for the underlying optimization package Pyomo. A useful way to start is downloading the open-source solvers, available for free from AMPL. For example, we recommend the solver CBC for mixed-integer problems (MILP) and the solver ipopt for non-linear (NLP) problems.

If you have access to a license of the powerful, commercial MILP-solvers Gurobi or CPLEX, you are encouraged to apply them to solve your *aristopy* model (provided your model is not non-linear). Academic users may be eligible to receive free licenses for both solvers. Please note that all installed solvers must be locatable by *aristopy*. Therefore, it is important to add the path to the solver executables to your system's environment variables.

CHAPTER

TWO

EXAMPLES

The following examples will guide you through the functionality of *aristopy*. The first example shows a straightforward application to introduce you to the basic model setup and the notation of *aristopy*. More advanced features are presented with a larger model in the second example. The last example model shows some functionality that is helpful when working with models that utilize solar components (solar thermal collectors or photovoltaic modules).

2.1 Model to get started

- File name: model_to_get_started.ipynb
- Last edited: 2020-06-24
- Created by: Stefan Bruche (TU Berlin)

```
import aristopy as ar

# Create basic energy system instance
es = ar.EnergySystem(
    number_of_time_steps=3, hours_per_time_step=1,
    interest_rate=0.05, economic_lifetime=20)

# Add a gas source, two different conversion units and sinks
gas_source = ar.Source(
    ensys=es, name='gas_source', commodity_cost=20, outlet=ar.Flow('Fuel'))

gas_boiler = ar.Conversion(
    ensys=es, name='gas_boiler', basic_variable='Heat',
    inlet=ar.Flow('Fuel', 'gas_source'), outlet=ar.Flow('Heat', 'heat_sink'),
    capacity_max=150, capex_per_capacity=60e3,
    user_expressions='Heat == 0.9 * Fuel')

chp_unit = ar.Conversion(
    ensys=es, name='chp_unit', basic_variable='Elec',
    inlet=ar.Flow('Fuel', 'gas_source'),
    outlet=[ar.Flow('Heat', 'heat_sink'), ar.Flow('Elec', 'elec_sink')],
    capacity_max=100, capex_per_capacity=600e3,
    user_expressions=['Heat == 0.5 * Fuel',
                      'Elec == 0.4 * Fuel'])

heat_sink = ar.Sink(
    ensys=es, name='heat_sink', inlet=ar.Flow('Heat'),
```

(continues on next page)

(continued from previous page)

```
    commodity_rate_fix=ar.Series('heat_demand', [100, 200, 150]))

elec_sink = ar.Sink(
    ensys=es, name='elec_sink', inlet=ar.Flow('Elec'), commodity_revenues=30)

# Run the optimization
es.optimize(solver='cbc', results_file='results.json')

# Plot some results
plotter = ar.Plotter('results.json')
plotter.plot_operation('heat_sink', 'Heat', lgd_pos='lower center',
                       bar_lw=0.5, ylabel='Thermal energy [MWh]')
plotter.plot_objective(lgd_pos='lower center')
```

2.1.1 Create *aristopy* model

First, we need to import the *aristopy* package. If the import fails, you might need to recheck the installation instructions.

[1]:
```
# Import the required packages (jupyter magic only required for jupyter notebooks)
%reload_ext autoreload
%autoreload 2
%matplotlib inline

import aristopy as ar
```

An *aristopy* model consists of an instance of the EnergySystem class and the added components. To create an energy system, we need to specify the number of considered time steps and the number of hours per time step. Additionally, the interest rate and the economic lifetime of the installed components are required to calculate the net present value (objective function value).

[2]:
```
# Create basic energy system instance
es = ar.EnergySystem(number_of_time_steps=3, hours_per_time_step=1,
                     interest_rate=0.05, economic_lifetime=20)
```

To instantiate a Component instance (Source, Sink, Conversion, Bus, Storage), we need to specify the EnergySystem instance, where it is added to and set a name for the component. Next, we add flows on the inlets and outlets. A Flow instance represents a connection point of a component and is used to create links with other components. Additionally, the flow introduces a commodity to the component and triggers the creation of an associated commodity variable (usually with the same name). The number of required or accepted inlet and outlet flows and component commodities depends on the component type (see table below). You can add multiple flows on an inlet or outlet for setting different commodities or linking components, by arranging them in a list.

Component type	Nbr. of inlet flows	Nbr. of outlet flows	Nbr. of commodities
Source	0	≥ 1	1
Sink	≥ 1	0	1
Conversion	≥ 1	≥ 1	≥ 1
Storage	≥ 1	≥ 1	1
Bus	≥ 1	≥ 1	1

[3]:
```
# Add a gas source
gas_source = ar.Source(ensys=es, name='gas_source', outlet=ar.Flow('Fuel'),
                       commodity_cost=20)
```

The conversion instances usually have different commodities on their inlets and outlets. That's why we need to specify the name of the basic variable for conversion components. This basic variable is used to restrict capacities, set operation rates, and calculate CAPEX and OPEX.

[4]:
```
# Add a gas boiler conversion unit
gas_boiler = ar.Conversion(ensys=es, name='gas_boiler',
                           basic_variable='Heat',
                           inlet=ar.Flow(commodity='Fuel', link='gas_source'),
                           outlet=ar.Flow('Heat', 'heat_sink'),
                           capacity_max=150, capex_per_capacity=60e3,
                           user_expressions='Heat == 0.9 * Fuel')
```

We can use the keyword argument **user_expressions** to specify commodity conversion rates, limit capacities, and set other internal component constraints manually. Here we can use the names (identifiers) of the commodity variables created by adding flows, and, if applicable, variables with standard names, e.g.:

- CAP - component capacity variable
- BI_EX - binary existence variable
- BI_OP - binary operation variable
- ... (see file utils.py in your aristopy directory)

The expressions are simply added as a list of strings. The options for mathematical operators are: `sum, sin, cos, exp, log, ==, >=, <=, **, *, /, +, -, (, )`. The indexes (sets) of the variables and parameters are processed automatically behind the scenes.

[5]:
```
# Add a CHP unit
chp_unit = ar.Conversion(ensys=es, name='chp_unit', basic_variable='Elec',
                         inlet=ar.Flow('Fuel', 'gas_source'),
                         outlet=[ar.Flow('Heat', 'heat_sink'), ar.Flow('Elec', 'elec_sink
↪')],
                         capacity_max=100, capex_per_capacity=600e3,
                         user_expressions=['Heat == 0.5 * Fuel',
                                           'Elec == 0.4 * Fuel'])
```

Time series data can be introduced as an aristopy Series instance and might be applied to set commodity rates, and time-dependent commodity cost or revenues, or generally for the scripting of user expressions.

[6]:
```
# Add a sink with fixed heat demand
heat_sink = ar.Sink(ensys=es, name='heat_sink', inlet=ar.Flow('Heat'),
                    commodity_rate_fix=ar.Series('heat_demand', [100, 200, 150]))

elec_sink = ar.Sink(ensys=es, name='elec_sink', inlet=ar.Flow('Elec'),
                    commodity_revenues=30)
```

Note:

Alternatively, we could use the *time_series_data* and *user_expressions* keyword arguments so set the required fixed commodity rate of the heat sink.

```
heat_sink = ar.Sink(ensys=es, name='heat_sink', inlet=ar.Flow('Heat'),
                    time_series_data=ar.Series('heat_demand', [100, 200, 150]),
                    user_expressions='Heat == heat_demand')
```

2.1.2 Run optimization

To run the optimization, we need to call the EnergySystem method *optimize*. The most important input to this method is the name of the applied solver. You have to ensure the solver is available on your machine and can be detected with this name. The solver output is suppressed for convenience in this notebook (*tee=False*). The results of the model run are written to a JSON-file with a specified name.

[7]:
```
es.optimize(solver='cbc', tee=False, results_file='results.json')
```

Basic information about the building and solving process of the optimzation model are stored in the Python dictionary *run_info* of the EnergySystem instane.

[8]:
```
es.run_info
```

[8]:
```
{'solver_name': 'cbc',
 'time_limit': None,
 'optimization_specs': '',
 'model_build_time': 0,
 'model_solve_time': 0,
 'upper_bound': 349147961.7,
 'lower_bound': 349147961.7,
 'sense': 'maximize',
 'solver_status': 'ok',
 'termination_condition': 'optimal'}
```

The pyomo ConcreteModel instance of the energy system can be accessed with the attribute *model*. All of the conventional pyomo functions can be applied here (e.g., pprint of the objective function).

[9]:
```
es.model.Obj.pprint()
```

```
Obj : Size=1, Index=None, Active=True
    Key  : Active : Sense    : Expression
    None :   True : maximize : -249.24420685079974*(gas_source.Fuel[0,0] + gas_source.
↪Fuel[0,1] + gas_source.Fuel[0,2])/0.00034246575342465754 - 60000.0*gas_boiler.CAP -␣
↪600000.0*chp_unit.CAP + 373.8663102761996*(elec_sink.Elec[0,0] + elec_sink.Elec[0,1] +␣
↪elec_sink.Elec[0,2])/0.00034246575342465754
```

The component variables and constraints are stored in separate pyomo Block models. They can be accessed via attribute block directly on the components. All components are also added to EnergySystem's dictionary components and can be reached with their specified name.

[10]:
```
gas_boiler.block.Heat.pprint()
```

```
Heat : Size=3, Index=time_set
    Key    : Lower : Value : Upper : Fixed : Stale : Domain
    (0, 0) :     0 :   0.0 :  None : False : False : NonNegativeReals
    (0, 1) :     0 :  75.0 :  None : False : False : NonNegativeReals
    (0, 2) :     0 :  25.0 :  None : False : False : NonNegativeReals
```

[11]:
```
# return dictionary of variable 'Elec' for component 'chp_unit'
es.components['chp_unit'].block.Elec.get_values()
```

[11]:
```
{(0, 0): 80.0, (0, 1): 100.0, (0, 2): 100.0}
```

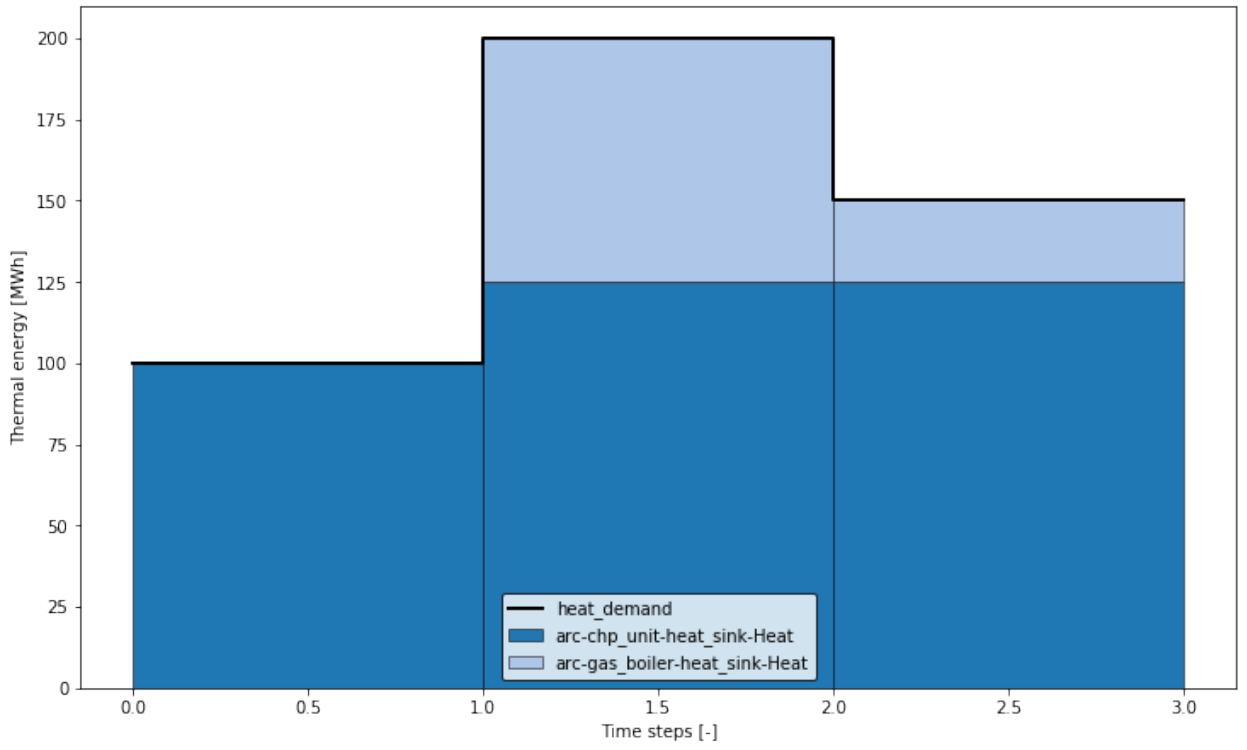
2.1.3 Plot results

The Plotter class is used to read the exported optimization results from the JSON-file and to provide basic plotting routines. Additional keyword arguments are available to customize the plotting output, e.g., set labels, figure size, legend position, etc. (see dictionary *props* of the Plotter class).

[12]:
```
# Create instance of Plotter class and read in file 'results.json'
plotter = ar.Plotter('results.json')
```

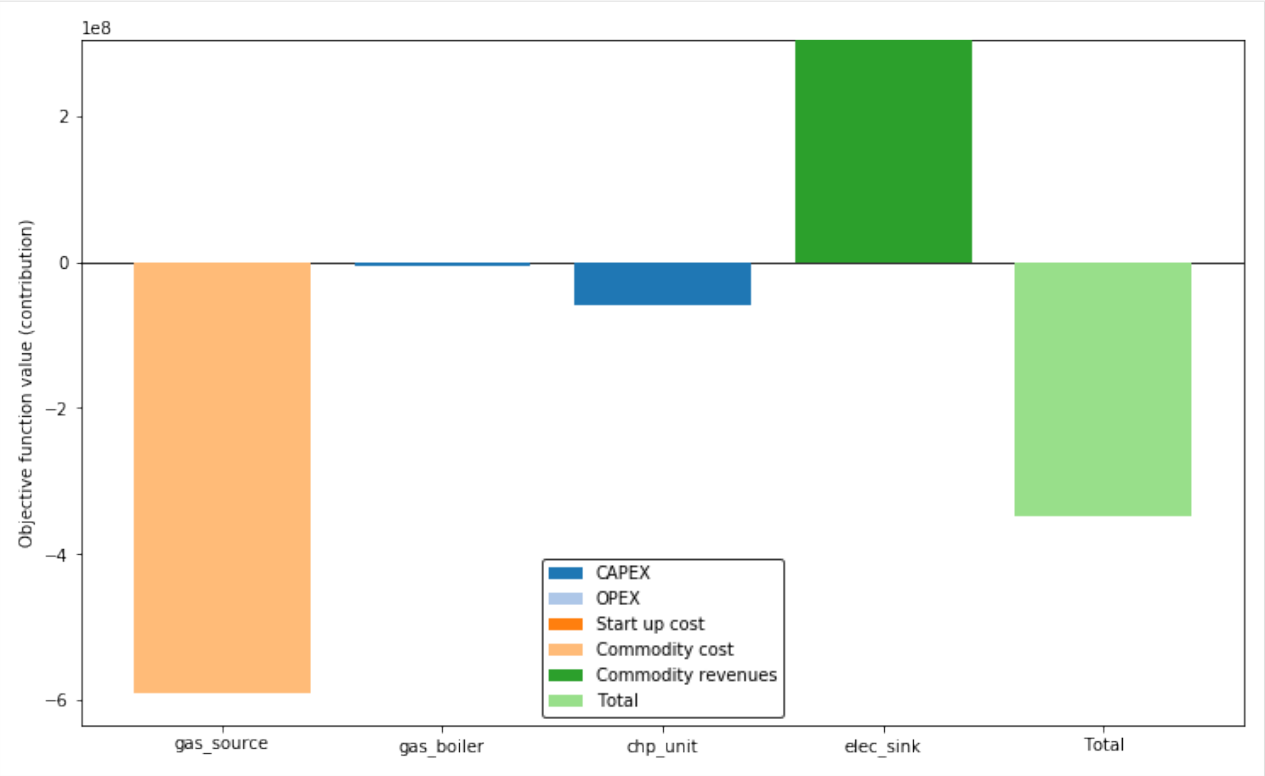
The method *plot_operation* returns a mixed bar and line plot that visualizes the operation of a component on the basis of a selected commodity.

[13]:
```
plotter.plot_operation('heat_sink', 'Heat', lgd_pos='lower center',
                       bar_lw=0.5, ylabel='Thermal energy [MWh]',
                       show_plot=True)
```



The method *plot_objective* returns a bar chart that summarizes the cost contributions of each component to the overall objective function value.

[14]:
```
plotter.plot_objective(lgd_pos='lower center', show_plot=True)
```



2.2 Advanced features

- File name: advanced_features.ipynb
- Last edited: 2020-06-25
- Created by: Stefan Bruche (TU Berlin)

```
import copy
import pandas as pd
import aristopy as ar

# Get data from csv-file
data = pd.read_csv('testdata.csv', sep=';', decimal='.', index_col=[0])

# USER-INPUT: "hpts" (hours per time step)
hpts = 4

# Aggregate data with "hours_per_time_step" for simplification of this notebook
c_elec = ar.utils.aggregate_time_series(data['c_elec [EUR/MWh]'], hpts, 'mean')
q_demand = ar.utils.aggregate_time_series(data['Q_demand [MWh]'], hpts, 'sum')

# Create energy system instance
es = ar.EnergySystem(
    number_of_time_steps=8760//hpts, hours_per_time_step=hpts,
    interest_rate=0.05, economic_lifetime=20, logging=ar.Logger(
        default_log_handler='file', default_log_level='INFO',
```

(continues on next page)

(continued from previous page)

```
        write_screen_output_to_logfile=True))

# Add a gas source, a heat and electricity sink and a heat bus for collecting
gas_source = ar.Source(
    ensys=es, name='gas_source', commodity_cost=20, outlet=ar.Flow('Fuel'))

heat_sink = ar.Sink(
    ensys=es, name='heat_sink', inlet=ar.Flow('Heat', link='heat_bus'),
    commodity_rate_fix=ar.Series('q_demand', q_demand))

elec_sink = ar.Sink(
    ensys=es, name='elec_sink', inlet=ar.Flow('Elec'),
    commodity_revenues=ar.Series('c_elec', c_elec))

heat_bus = ar.Bus(
    ensys=es, name='heat_bus',
    inlet=ar.Flow('Heat', var_name='Heat_inlet'),
    outlet=ar.Flow('Heat', var_name='Heat_outlet'))

gas_boiler = ar.Conversion(
    ensys=es, name='gas_boiler', basic_variable='Heat',
    inlet=ar.Flow('Fuel', 'gas_source'), outlet=ar.Flow('Heat', 'heat_bus'),
    capacity_max=250, capex_per_capacity=60e3,
    user_expressions='Heat == 0.9 * Fuel')

# Add a group of 5 gas engine units with fixed size
gas_engine = ar.Conversion(
    ensys=es, name='gas_engine', basic_variable='Elec',
    inlet=ar.Flow('Fuel', 'gas_source'),
    outlet=[ar.Flow('Heat', 'heat_bus'), ar.Flow('Elec', 'elec_sink')],
    has_existence_binary_var=True, has_operation_binary_var=True,
    additional_vars=ar.Var('Load', ub=1), scalar_params={'dt': hpts},
    capacity=20, min_load_rel=0.2,
    user_expressions=['Load <= BI_OP',
                      'Elec == (20 * Load) * dt',
                      'Heat <= (17 * Load + 2.5 * BI_OP) * dt',
                      'Fuel == (37 * Load + 7 * BI_OP) * dt'],
    capex_per_capacity=1000e3, opex_per_capacity=12.5e3, opex_operation=6,
    instances_in_group=5)

# Add a heat storage component
heat_storage = ar.Storage(
    ensys=es, name='heat_storage',
    inlet=ar.Flow('Heat', link='heat_bus', var_name='Heat_charge'),
    outlet=ar.Flow('Heat', link='heat_bus', var_name='Heat_discharge'),
    has_existence_binary_var=True,
    maximal_module_number=5, capacity_per_module=50,  # max: 5 * 50 MWh
    capex_per_capacity=12e3, opex_per_capacity=0.015*12e3,  # 12,000 €/MWh CAPEX
    self_discharge=0.01, charge_rate=1/6, discharge_rate=1/6,
    opex_operation=1e-9, use_inter_period_formulation=False)

# Use a backdoor to the pyomo model instance and add a user-defined constraint
```

(continues on next page)

(continued from previous page)

```
# specifying the installed nominal thermal capacity needs to be at least 220 MW.
def minimal_installed_capacity_heat(ensys, m):
    expr = 0
    for comp in ensys.components.values():
        if comp.name == 'gas_boiler':
            expr += comp.block.CAP
        elif comp.group_name == 'gas_engine':
            expr += 19.5 * comp.block.BI_EX
    return expr >= 220
es.add_constraint(name='min_cap_heat', has_time_set=False,
                  rule=minimal_installed_capacity_heat)


# Create a deepcopy of the original (full scale) model instance for later use
original_model = copy.deepcopy(es)


# Conventional optimization:
# ~~~~~~~~~~~~~~~~~~~~~~~~~~
# Run the optimization of the original model instance
es.optimize(solver='gurobi', time_limit=300, optimization_specs='mipgap=5e-3')


# Time series clustering:
# ~~~~~~~~~~~~~~~~~~~~~~~
# Cluster time series data to 20 typical days.
es.cluster(number_of_typical_periods=20,
           number_of_time_steps_per_period=24//hpts)

# Use aggregated data to run the optimization
es.optimize(use_clustered_data=True)

# Plot results for a single period and scaled to full year with hourly resolution
plotter = ar.Plotter('results.json')
plotter.plot_operation('heat_storage', 'Heat', ylabel='Thermal energy [MWh]',
                       file_name='heat_storage', plot_single_period_with_index=7)
plotter.plot_operation('heat_bus', 'Heat', ylabel='Thermal energy [MWh]',
                       file_name='heat_bus', scale_to_hourly_resolution=True)


# Two-stage model run:
# ~~~~~~~~~~~~~~~~~~~~
# Construct (declare) a persistent model of the original (full scale) instance
original_model.declare_model(use_clustered_data=False, declare_persistent=True)

# Cluster time series data, declare the model and relax the integrality of the
# binary operation variables
es.cluster(number_of_typical_periods=20,
           number_of_time_steps_per_period=24 // hpts)
es.declare_model(use_clustered_data=True)
es.relax_integrality(include_time_dependent=True,
                     include_existence=False, include_modules=False)
```

(continues on next page)

(continued from previous page)

```
# Iteratively run the model (1st stage and 2nd stage)
for icc in range(3):

    # Run optimization with clustered data
    es.optimize(use_clustered_data=True, declare_model=False,
                tee=False, results_file=f'1st_stage_results_{icc}.json')

    # Export the configuration to a DataFrame and store it in an Excel-File
    config = es.export_component_configuration(
        write_to_excel=True, file_name=f'1st_stage_config_{icc}.xlsx')

    # Import the optimal configuration and fix design variables
    original_model.import_component_configuration(config)

    # Run the already declared persistent, full scale model
    original_model.optimize(declare_model=False, tee=False,
                            results_file=f'2nd_stage_results_{icc}.json')

    # Reset variables to their original (unfixed) values
    original_model.reset_component_variables()

    # Add an integer-cut constraint for the gas engines to the aggregated model
    # => Exclude previously found optimal design from the solution space
    es.add_design_integer_cut_constraint(which_instances=['gas_engine'])
```

2.2.1 Data and model

Read the input data from a csv-file for one year in hourly resolution (8760 time steps) with pandas.

[1]:
```
# Import the required packages (jupyter magic only required for jupyter notebooks)
%reload_ext autoreload
%autoreload 2
%matplotlib inline

import copy
import pandas as pd
import aristopy as ar

# Get data from csv-file
data = pd.read_csv('testdata.csv', sep=';', decimal='.', index_col=[0])
```

[2]:
```
# print the first 8 rows
data.head(8)
```

[2]:

	c_elec [EUR/MWh]	Q_demand [MWh]
01.01.2018 00:00	-5.27	51.828932
01.01.2018 01:00	-29.99	51.123716
01.01.2018 02:00	-56.65	49.202127
01.01.2018 03:00	-63.14	49.060032
01.01.2018 04:00	-64.62	52.669893

(continues on next page)

(continued from previous page)

```
01.01.2018 05:00            -67.00       59.055207
01.01.2018 06:00            -72.54       64.771925
01.01.2018 07:00            -76.01       66.424269
```

Aggregate the data for simplification purposes in this notebook by summing/averaging over 4 hours. There is a built-in function in the file utils.py to assist in completing this task.

[3]:
```
# USER-INPUT: "hpts" (hours per time step)
hpts = 4

# Aggregate data with "hours_per_time_step" for simplification of this notebook
c_elec = ar.utils.aggregate_time_series(data['c_elec [EUR/MWh]'], hpts, 'mean')
q_demand = ar.utils.aggregate_time_series(data['Q_demand [MWh]'], hpts, 'sum')

# Show aggregated data on screen
c_elec, q_demand
```

[3]:
```
(array([-38.7625, -70.0425, -64.1025, ...,  61.3   ,  64.1375,  43.5775]),
 array([201.21480811, 242.92129394, 306.69530448, ..., 432.8322987 ,
        463.4687946 , 381.27112154]))
```

As in the first example, first, an energy system instance must be created as the main model container. This time an additional logger is added to store information about the modeling and solving process in a log-file. Moreover, a gas source and electricity and heat sinks are added.

[ ]:
```
# Create energy system instance
es = ar.EnergySystem(
    number_of_time_steps=8760//hpts, hours_per_time_step=hpts,
    interest_rate=0.05, economic_lifetime=20, logging=ar.Logger(
        default_log_handler='file', default_log_level='INFO',
        write_screen_output_to_logfile=True))

# Add a gas source, a heat and electricity sink and a heat bus for collecting
gas_source = ar.Source(
    ensys=es, name='gas_source', commodity_cost=20, outlet=ar.Flow('Fuel'))

heat_sink = ar.Sink(
    ensys=es, name='heat_sink', inlet=ar.Flow('Heat', link='heat_bus'),
    commodity_rate_fix=ar.Series('q_demand', q_demand))

elec_sink = ar.Sink(
    ensys=es, name='elec_sink', inlet=ar.Flow('Elec'),
    commodity_revenues=ar.Series('c_elec', c_elec))
```

The bus component collects all heat flows at the inlet and transports them (with or without losses) to the outlet.

Note:

The same commodity is present at the inlet and the outlet. Therefore, different names for the created variables must be specified in the corresponding flows.

[5]:
```
heat_bus = ar.Bus(
    ensys=es, name='heat_bus',
    inlet=ar.Flow('Heat', var_name='Heat_inlet'),
    outlet=ar.Flow('Heat', var_name='Heat_outlet'))

gas_boiler = ar.Conversion(
    ensys=es, name='gas_boiler', basic_variable='Heat',
    inlet=ar.Flow('Fuel', 'gas_source'), outlet=ar.Flow('Heat', 'heat_bus'),
    capacity_max=250, capex_per_capacity=60e3,
    user_expressions='Heat == 0.9 * Fuel')
```

Next, we add five gas engine instances with identical specifications that are simultaneously created and arranged in a group (see "instances_in_group"). The five components can work independently, but have an order for their binary existence and operation variables (see API of the *Conversion class*).

[6]:
```
# Add maximal 5 gas engine units with fixed size
gas_engine = ar.Conversion(
    ensys=es, name='gas_engine', basic_variable='Elec',
    inlet=ar.Flow('Fuel', 'gas_source'),
    outlet=[ar.Flow('Heat', 'heat_bus'), ar.Flow('Elec', 'elec_sink')],
    has_existence_binary_var=True, has_operation_binary_var=True,
    additional_vars=ar.Var('Load', ub=1), scalar_params={'dt': hpts},
    capacity=20, min_load_rel=0.2,
    user_expressions=['Load <= BI_OP',
                      'Elec == (20 * Load) * dt',
                      'Heat <= (17 * Load + 2.5 * BI_OP) * dt',
                      'Fuel == (37 * Load + 7 * BI_OP) * dt'],
    capex_per_capacity=1000e3, opex_per_capacity=12.5e3, opex_operation=6,
    instances_in_group=5)
```

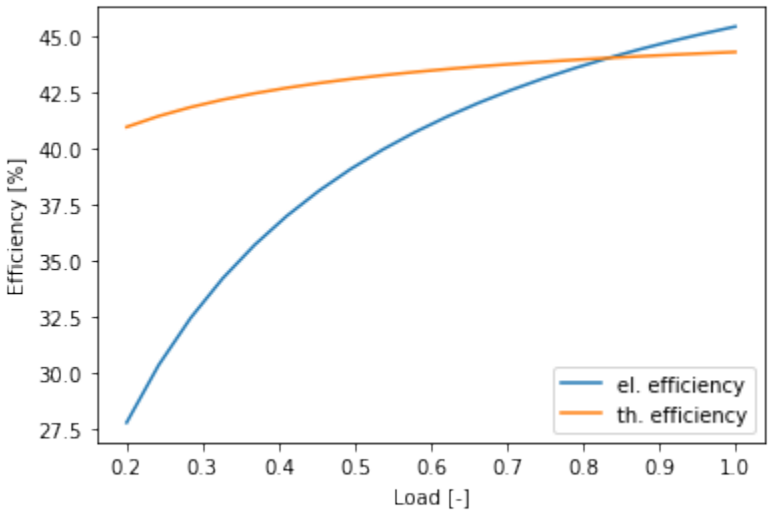
The performance modeling in the user expressions of the gas engines involves using binary operating variables (default name "BI_OP" is defined in file utils.py). Due to the y-axis intercept, the resulting efficiency curves show non-linear characteristics.

Note:

If required, we could also introduce time-varying parameters in the user-expressions to describe the dependency of the conversion rates on, e.g., ambient conditions.

---

[7]:
```
import matplotlib.pyplot as plt
import numpy as np

idx = np.linspace(0.2, 1, 20)
plt.plot(idx, (20*idx)/(37*idx+7)*100, label='el. efficiency')
plt.plot(idx, (17*idx+2.5)/(37*idx+7)*100, label='th. efficiency')
plt.xlabel('Load [-]'), plt.ylabel('Efficiency [%]'), plt.legend(loc=4)
plt.show()
```



A heat storage component is also included in the design. The storage tank has a modular design and can consist of up to 5 modules of 50 MWh. It should be noted that, similar to the bus component, different variable names must be specified for the inlet and outlet flows. The small value for the argument "opex_operation" is set to prevent simultaneous loading and unloading of the storage. For all other keyword arguments, please consult the documentation of the *storage class API*.

[8]:
```
# Add a heat storage component
heat_storage = ar.Storage(
    ensys=es, name='heat_storage',
    inlet=ar.Flow('Heat', link='heat_bus', var_name='Heat_charge'),
    outlet=ar.Flow('Heat', link='heat_bus', var_name='Heat_discharge'),
    has_existence_binary_var=True,
    maximal_module_number=5, capacity_per_module=50,  # max: 5 * 50 MWh
    capex_per_capacity=12e3, opex_per_capacity=0.015*12e3,  # 12,000 €/MWh CAPEX
    self_discharge=0.01, charge_rate=1/6, discharge_rate=1/6,
    opex_operation=1e-9, use_inter_period_formulation=False)
```

Finally, we introduce a constraint that the total installed thermal capacity of the gas boiler and all gas engines must be at least 220 MW. To add this condition to the model, we use a back door to the main pyomo model instance, accessible via the method `add_constraint` of the energy system. It must be determined whether the constraint is time-dependent (or has a different set), and a function must be assigned as the constraint rule. This function needs to have at least two arguments. The first one represents the energy system instance; the second one is the pyomo model instance. For a time-dependent constraint, we would need to add two more arguments, representing period (p) and time-step (t). Furthermore, *aristopy* offers the possibility to add additional variables and objective function contributions to the main pyomo model instance. The functions `add_variable` and `add_objective_function_contribution` are available for this purpose. Please consult the API of the *EnergySystem class* for further information.

[9]:
```
# Use a backdoor to the pyomo model instance and add a user-defined constraint
# specifying the installed nominal thermal capacity needs to be at least 220 MW.
def minimal_installed_capacity_heat(ensys, m):
    expr = 0
    for comp in ensys.components.values():
        if comp.name == 'gas_boiler':
            expr += comp.block.CAP
        elif comp.group_name == 'gas_engine':
            expr += 19.5 * comp.block.BI_EX
```

(continues on next page)

(continued from previous page)

```
    return expr >= 220
es.add_constraint(name='min_cap_heat', has_time_set=False,
                  rule=minimal_installed_capacity_heat)
```

Before we pass the developed model to a solver, we create a copy by using the method `deepcopy` of the copy-package. We will use this original model instance later.

[10]:
```
# Create a deepcopy of the original (full scale) model instance for later use
original_model = copy.deepcopy(es)
```

2.2.2 Conventional optimization

As in the first example, the model can simply be by calling the method `optimize` of the energy system instance. To control the solving process, we add a time limitation in seconds and a requested relative MIP-gap to the solver Gurobi.

[11]:
```
# Run the optimization of the original model instance
es.optimize(solver='gurobi', time_limit=300, optimization_specs='mipgap=1e-3',
            results_file='full_scale_results.json', tee=False)
```

[12]:
```
# show basic information about the model building and solving process
es.run_info
```

[12]:
```
{'solver_name': 'gurobi',
 'time_limit': 300,
 'optimization_specs': 'mipgap=1e-3',
 'model_build_time': 4,
 'model_solve_time': 308,
 'upper_bound': -163634109.14349702,
 'lower_bound': -163920996.8127097,
 'sense': 'maximize',
 'solver_status': 'aborted',
 'termination_condition': 'maxTimeLimit'}
```

The optimal design solution of the original (full scale) model instance can be accessed with the `export_component_configuration` method and is stated in the table below. The objective function contributions of the individual components are shown in the following figure.

[13]:
```
es.export_component_configuration()
```

[13]:

	gas_source	heat_sink	elec_sink	heat_bus	gas_boiler	gas_engine_1	\
BI_EX	None	None	None	None	None	1	
BI_MODULE_EX	None	None	None	None	None	None	
CAP	None	None	None	None	142	20	

	gas_engine_2	gas_engine_3	gas_engine_4	gas_engine_5	\
BI_EX	1	1	1	-0	
BI_MODULE_EX	None	None	None	None	
CAP	20	20	20	0	

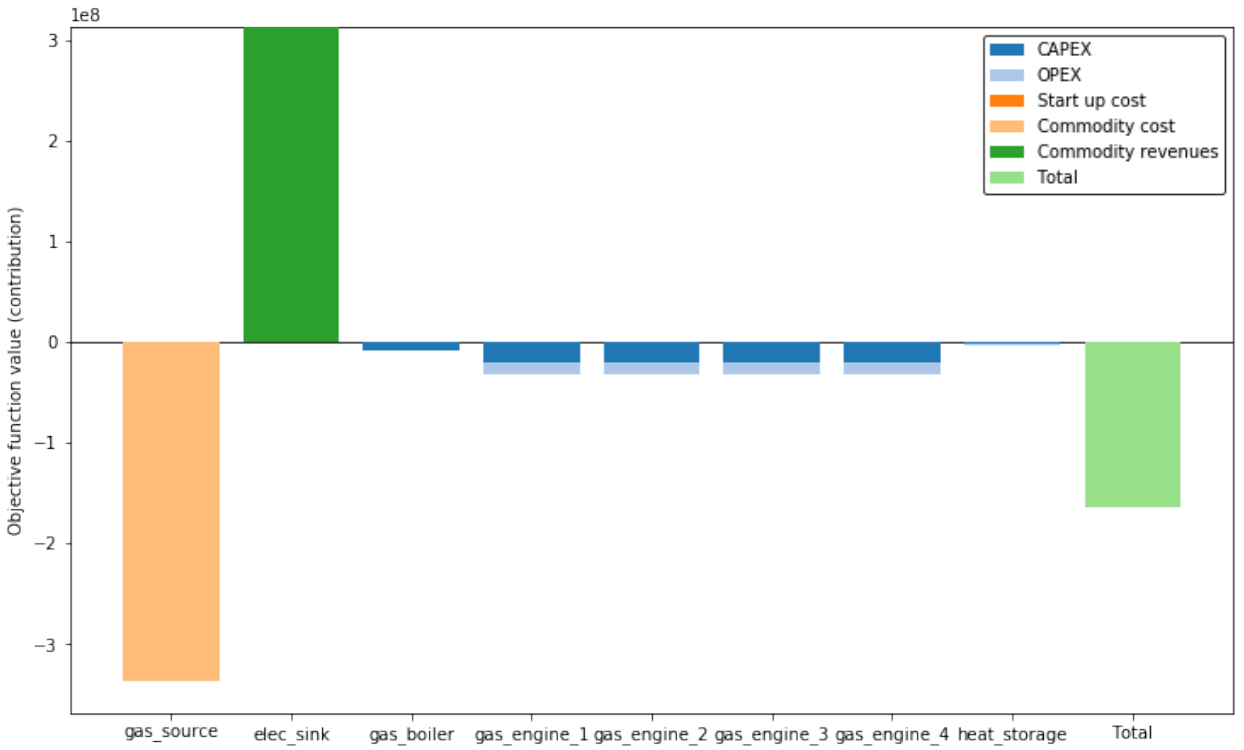
	heat_storage
BI_EX	1

(continues on next page)

(continued from previous page)

```
BI_MODULE_EX  {1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0, 5: -0.0}
CAP                                                  200
```

[14]:
```
ar.Plotter('full_scale_results.json').plot_objective(show_plot=True)
```



2.2.3 Time series clustering

To reduce the number of variables and to speed up the calculation, we can perform the optimization for only a few typical periods. To do this, we first call the `cluster` method and break the time series data into elements of equal length and combine similar items into clusters. In the example below, 20 typical days are formed from the input data, and the optimization is performed with these aggregated data ("use_clustered_data" is True).

[15]:
```
# Cluster time series data to 20 typical days.
es.cluster(number_of_typical_periods=20,
           number_of_time_steps_per_period=24//hpts)

# Use clustered data to run the optimization
es.optimize(use_clustered_data=True, tee=False, results_file='clustered_results.json')
```

The required time to build and solve the model can be reduced significantly by using clustered input data. The optimal design solution found is comparable to the one of the original model.

[16]:
```
es.run_info
```

[16]:
```
{'solver_name': 'gurobi',
 'time_limit': None,
 'optimization_specs': '',
```

(continues on next page)

(continued from previous page)

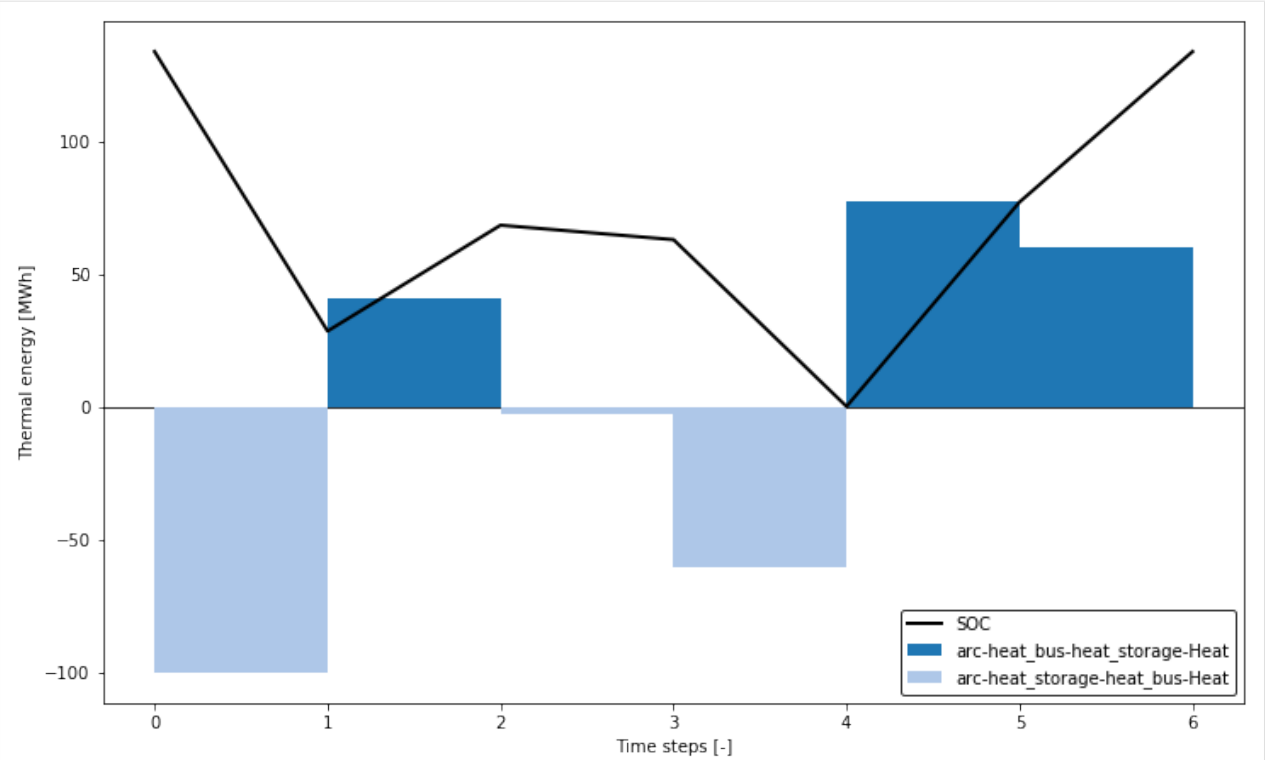
```
 'model_build_time': 0,
 'model_solve_time': 2,
 'upper_bound': -170278358.02702567,
 'lower_bound': -170278358.02702567,
 'sense': 'maximize',
 'solver_status': 'ok',
 'termination_condition': 'optimal'}
```

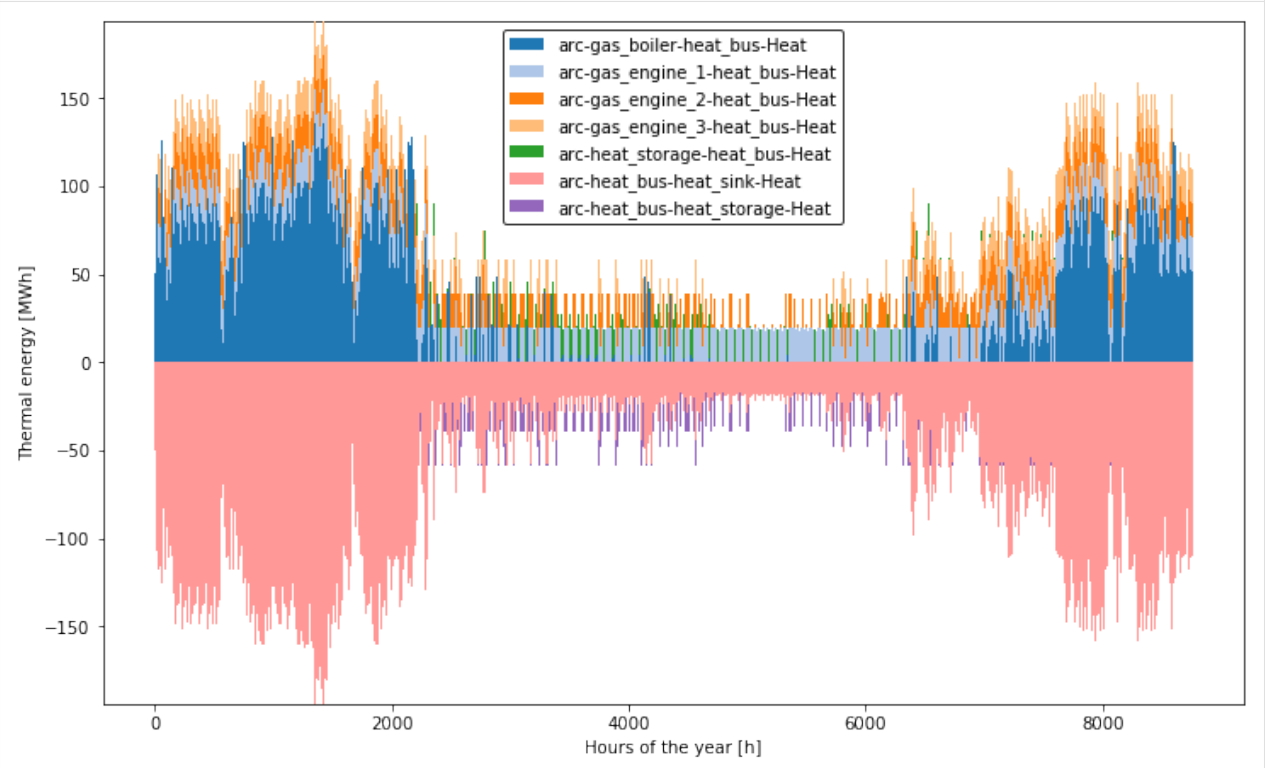
[17]:
```
es.export_component_configuration()
```

[17]:
```
             gas_source heat_sink elec_sink heat_bus gas_boiler gas_engine_1  \
BI_EX              None      None      None     None       None            1
BI_MODULE_EX       None      None      None     None       None         None
CAP                None      None      None     None      161.5           20

             gas_engine_2 gas_engine_3 gas_engine_4 gas_engine_5  \
BI_EX                   1            1           -0           -0
BI_MODULE_EX         None         None         None         None
CAP                    20           20           -0           -0

                                          heat_storage
BI_EX                                                1
BI_MODULE_EX  {1: 1.0, 2: 1.0, 3: 1.0, 4: -0.0, 5: -0.0}
CAP                                                150
```

The following figures show the state of charge (SOC) of the heat storage tank and the associated loading and unloading operations for a selected typical period. Besides, the heat flows, entering and leaving the heat bus component are plotted scaled to a full year cycle.

[18]:
```
plotter = ar.Plotter('clustered_results.json')
plotter.plot_operation('heat_storage', 'Heat', ylabel='Thermal energy [MWh]',
                       file_name='heat_storage', plot_single_period_with_index=7,
                       show_plot=True)
plotter.plot_operation('heat_bus', 'Heat', ylabel='Thermal energy [MWh]',
                       file_name='heat_bus', scale_to_hourly_resolution=True,
                       show_plot=True)
```


Thermal energy [MWh]
100
50
0
−50
−100
0
1
2
3
4
5
6
Time steps [-]
SOC
arc-heat_bus-heat_storage-Heat
arc-heat_storage-heat_bus-Heat


Thermal energy [MWh]
150
100
50
0
−50
−100
−150
0
2000
4000
6000
8000
Hours of the year [h]
arc-gas_boiler-heat_bus-Heat
arc-gas_engine_1-heat_bus-Heat
arc-gas_engine_2-heat_bus-Heat
arc-gas_engine_3-heat_bus-Heat
arc-heat_storage-heat_bus-Heat
arc-heat_bus-heat_sink-Heat
arc-heat_bus-heat_storage-Heat

2.2.4 Two-stage model run

The last part of this example shows the possibility of dividing the original problem into two levels to reduce computation time and enable the fast generation of several high-quality design solution candidates. The developers of *aristopy* have published a comparable method in this paper. In the first optimization stage, a simplified problem is considered, and a design candidate is generated. The simplification is achieved by the aggregation of time series data and the relaxation of binary operation variables. In the second step, the design candidate's objective function value is determined for the original problem by considering the full-scale model.

Building a large optimization model in Pyomo can take a considerable time. Therefore Pyomo offers the possibility to create so-called persistent solvers (only for Gurobi and CPLEX). This solver instance is created only once, and the model is added via its persistent interface. Later, this model can be solved various times without the time-consuming building phase. The flag "declare_persistent" must be set to True during model declaration to exploit these persistent models in *aristopy*. Minor changes can be added to the persistent solver at a later stage. More information about the persistent solver interface of Pyomo can be found here.

[19]:
```
# Construct (declare) a persistent model of the original (full scale) instance
original_model.declare_model(use_clustered_data=False, declare_persistent=True)
```

In the following steps, the cluster function is called for the energy system model (not the original/full-scale model!), and the model is declared with aggregated data. Then the binary operating variables (ON/OFF) are relaxed so that they can take on values between zero and one. The existence variables remain binary.

[20]:
```
# Cluster time series data, declare the model and relax the integrality of the
# binary operation variables
es.cluster(number_of_typical_periods=20,
           number_of_time_steps_per_period=24//hpts)
es.declare_model(use_clustered_data=True)
es.relax_integrality(include_time_dependent=True,
                     include_existence=False, include_modules=False)
```

Subsequently, several optimization problems are iteratively solved, and the results are stored in a pandas DataFrame for further comparison. In the first step, the problem is solved with clustered data (model is not declared again!). The configuration (existence and capacity of the components) is imported into the full-scale problem. Now, this original problem is solved with a fixed design as a pure dispatch model. In the last step, an integer cut constraint is introduced in the aggregated model. Thus, the current design solution (the combination of binary variables) is excluded from the solution space, and another design solution must be generated in the next model run.

[21]:
```
# Create a number of integer-cuts
nbr_of_icc = 5

# Store some results in a DataFrame to compare solutions
results = pd.DataFrame(index=range(nbr_of_icc), columns=[
    'Obj. value [Mio.€]', 'Nbr. of engines [-]',
    'Capacity storage [MWh]', 'Capacity boiler [MW]',
    '1st stage time [s]', '2nd stage time [s]'])

for icc in range(nbr_of_icc):

    # Run optimization with clustered data
    es.optimize(use_clustered_data=True, declare_model=False,
                tee=False, results_file=f'1st_stage_results_{icc}.json')

    # Export the configuration to a DataFrame and store it in an Excel-File
```

(continues on next page)

(continued from previous page)

```
    config = es.export_component_configuration(
        write_to_excel=True, file_name=f'1st_stage_config_{icc}.xlsx')

    # Import the optimal configuration and fix design variables
    original_model.import_component_configuration(config)

    # Run the already declared persistent, full scale model
    original_model.optimize(declare_model=False, tee=False,
                            results_file=f'2nd_stage_results_{icc}.json')

    # Reset variables to their original (unfixed) values
    original_model.reset_component_variables()

    # Add an integer-cut constraint for the gas engines to the aggregated model
    # => Exclude previously found optimal design from the solution space
    es.add_design_integer_cut_constraint(which_instances=['gas_engine'])


    results['Obj. value [Mio.€]'].loc[icc] = original_model.run_info['lower_bound'] / 1e6
    results['Nbr. of engines [-]'].loc[icc] = sum(comp.block.BI_EX.value
                                                  for comp in es.components.values()
                                                  if comp.group_name == 'gas_engine')
    results['Capacity storage [MWh]'].loc[icc] = config['heat_storage'].loc['CAP']
    results['Capacity boiler [MW]'].loc[icc] = config['gas_boiler'].loc['CAP']
    results['1st stage time [s]'].loc[icc] = es.run_info['model_solve_time']
    results['2nd stage time [s]'].loc[icc] = original_model.run_info['model_solve_time']
```

The following summary of the results shows that a single iteration can be performed much faster by splitting it into two optimization levels. The detected solutions are comparable to the globally optimal solution of the original problem (see above). In addition, further design candidates are evaluated, and a deeper understanding of the present model is gained.

[22]:
```
# print the results on the screen
results
```

[22]:

	Obj. value [Mio.€]	Nbr. of engines [-]	Capacity storage [MWh]	Capacity boiler [MW]	1st stage time [s]	2nd stage time [s]
0	-164.202	3	100	161.5	0	11
1	-166.424	2	100	181	0	4
2	-163.98	4	150	142	0	36
3	-172.912	1	0	200.5	0	1
4	-166.653	5	150	122.5	1	94

2.3 Solar components

- File name: solar_components.ipynb
- Last edited: 2020-06-30
- Created by: Stefan Bruche (TU Berlin)

Note:

The solar classes require the availability of the Python module *pvlib*. The module is not provided with the standard installation of *aristopy*. If you want to use the solar classes, consider installing the module in your current environment, e.g. via:

```
>> pip install pvlib
```

For further information and an installation guide, users are referred to the pvlib documentation.

```
import pandas as pd
import aristopy as ar

# Get data from csv-file
data = pd.read_csv('testdata.csv', sep=';', decimal='.', index_col=[0])

# Convert index to type DateTimeIndex and add required information about the time zone.
data.index = pd.to_datetime(
    data.index, format="%d.%m.%Y %H:%M").tz_localize(tz='UTC')

# Create a SolarData class instance with global (GHI) and diffuse (DHI)
# horizontal irradiation data for the location Potsdam/Germany.
solar = ar.SolarData(ghi=data['GHI [W/m2]'], dhi=data['DHI [W/m2]'],
                     latitude=52.3822, longitude=13.0622, altitude=81)


# Photovoltaic
# ------------
# Calculate the direct normal irradiation (DNI) from GHI, DHI and the solar positions
df_solar = solar.get_irradiance_dataframe()

# Append ambient temperature data (and wind speed if available)
df_solar['temp_air'] = data['T_amb [C]']

# Create a PV System (consisting of a module and an inverter).
pv_system = ar.PVSystem(module='Canadian_Solar_Inc__CS6X_300P',
                        inverter='Canadian_Solar_Inc___CSI_60KTL_CT__480V_')

# Calculate the feed-in of the PV system for specified conditions at a site
# (irradiation, temperature, collector tilt and azimuth).
pv_feed_in = pv_system.get_feedin(
    weather=df_solar, location=solar.location,
    surface_tilt=30, surface_azimuth=180,  # South
    modules_per_string=20, strings_per_inverter=10, scaling='peak_power')  # [W/Wp]
```

(continues on next page)

(continued from previous page)

```
# Solar thermal collector
# -----------------------
# Calculate the irradiance components on the solar-thermal collector array.
poa = solar.get_plane_of_array_irradiance(surface_tilt=45, surface_azimuth=180)

# Set up a solar collector array (type: evacuated tube collectors).
solar_collector_data = ar.SolarThermalCollector(
    optical_efficiency=0.80, thermal_loss_parameter_1=1.1,
    thermal_loss_parameter_2=0.008, irradiance_data=poa['poa_global'],
    t_ambient=data['T_amb [C]'], t_collector_in=60, t_collector_out=90)

# Calculate heat output of the solar-thermal collector
solar_heat = solar_collector_data.get_collector_heat_output()


# Solar components in the model
# -----------------------------
# Create energy system instance
es = ar.EnergySystem()

# Add a photovoltaic component (electricity source)
pv = ar.Source(
    ensys=es, name='pv', outlet=ar.Flow('Elec', 'elec_sink'),
    time_series_data=ar.Series('pv_feed_in', pv_feed_in),  # [MW/MWp]
    capacity=100, capex_per_capacity=700e3, opex_per_capacity=0.025*700e3,  # [MWp]
    user_expressions='Elec == CAP * pv_feed_in')

# Add a Solar-thermal collector component (heat source)
solar_collector = ar.Source(
    ensys=es, name='solar_collector', outlet=ar.Flow('Heat', 'heat_sink'),
    basic_variable='Area', additional_vars=ar.Var('Area', has_time_set=False),
    time_series_data=ar.Series('collector_feedin', solar_heat / 1e6),  # [MW/m²]
    user_expressions='Heat == collector_feedin * Area',
    capacity=1e5, capex_per_capacity=300, opex_per_capacity=300*0.01)  # [m²], [EUR/m²]

# Add sinks for electricity and heat
elec_sink = ar.Sink(ensys=es, name='elec_sink', inlet=ar.Flow('Elec'))
heat_sink = ar.Sink(ensys=es, name='heat_sink', inlet=ar.Flow('Heat'))

# Run the optimization
es.optimize()
```

2.3.1 SolarData class

Read the input data from a csv-file for one year in hourly resolution (8760 time steps) with pandas.

[1]:
```
# Import the required packages (jupyter magic only required for jupyter notebooks)
%reload_ext autoreload
%autoreload 2
%matplotlib inline

import pandas as pd
import matplotlib.pyplot as plt
import aristopy as ar

# Get data from csv-file
data = pd.read_csv('testdata.csv', sep=';', decimal='.', index_col=[0])
```

The index of the solar data needs be of type DateTimeIndex and must contain information about the time zone. The Data from the reference "Deutscher Wetterdienst" (DWD) is provided with a UTC timestamp. The same is true for the Photovoltaic Geographical Information System (PVGIS) data of the JRC. So, we convert the index to a datetime and localize it to the UTC time zone. Alternatively, we could use the pandas method `date_range` to create a new index with time zone information included.

[2]:
```
data.index = pd.to_datetime(
    data.index, format="%d.%m.%Y %H:%M").tz_localize(tz='UTC')
```

[3]:
```
# print the time zone of the data index
data.index.tz
```

[3]:
```
<UTC>
```

[4]:
```
# print first rows
data.head(3)
```

[4]:

	T_amb [C]	DHI [W/m2]	GHI [W/m2]
2018-01-01 00:00:00+00:00	11.3	0.0	0.0
2018-01-01 01:00:00+00:00	10.9	0.0	0.0
2018-01-01 02:00:00+00:00	11.1	0.0	0.0

A SolarData class instance is created with provided specifications for the location and irradiance data. The global (GHI) and diffuse (DHI) horizontal irradiation time series are required input arguments. The direct normal (beam) irradiation (DNI) can also be specified or is internally calculated based on GHI, DHI, and solar positions. The selected location is Potsdam/Germany. Later, the SolarData instance is used to calculate and return values for all irradiance time series `get_irradiance_dataframe`, and the plane of array irradiance (POA) `get_plane_of_array_irradiance`.

[5]:
```
solar = ar.SolarData(ghi=data['GHI [W/m2]'], dhi=data['DHI [W/m2]'],
                     latitude=52.3822, longitude=13.0622, altitude=81)
```

2.3.2 PVSystem class

First, we calculate the direct normal irradiation (DNI) from GHI, DHI, and solar positions of the respective location, and return the data in a pandas DataFrame.

[6]:
```
df_solar = solar.get_irradiance_dataframe()
```

If available, ambient temperature and wind speed data can be appended to the DataFrame. In this case, *pvlib* requires the column names: 'temp_air' and 'wind_speed'. This step is optional. Default values 20°C and 0 m/s are used, if no values are provided.

[7]:
```
df_solar['temp_air'] = data['T_amb [C]']
```

[8]:
```
# Print data of the first day
df_solar.iloc[7:17]
```

[8]:

	ghi	dhi	dni	temp_air
2018-01-01 07:00:00+00:00	0.000000	0.000000	-0.000000	8.7
2018-01-01 08:00:00+00:00	25.000000	22.222222	38.988546	7.8
2018-01-01 09:00:00+00:00	88.888889	58.333333	185.767900	7.7
2018-01-01 10:00:00+00:00	147.222222	80.555556	293.940748	8.0
2018-01-01 11:00:00+00:00	205.555556	72.222222	526.992007	8.4
2018-01-01 12:00:00+00:00	180.555556	97.222222	345.532381	9.1
2018-01-01 13:00:00+00:00	66.666667	61.111111	28.910901	8.6
2018-01-01 14:00:00+00:00	58.333333	44.444444	126.548503	7.9
2018-01-01 15:00:00+00:00	22.222222	19.444444	0.000000	7.3
2018-01-01 16:00:00+00:00	0.000000	0.000000	-0.000000	7.5

The next step is to create an instance of class PVSystem by selecting a type module and inverter from the pvlib's database. The full database currently consists of more than 20,000 modules and 3,000 inverters. To see the database you can either go through the CSV-files in the "data" directory of your pvlib installation, or use the method `retrieve_sam`.

[9]:
```
import pvlib

pvlib.pvsystem.retrieve_sam(name='cecmod')  # for inverters: name='cecinverter'
```

[9]:

	A10Green_Technology_A10J_S72_175	A10Green_Technology_A10J_S72_180	\
Technology	Mono-c-Si	Mono-c-Si	
Bifacial	0	0	
STC	175.091	179.928	
PTC	151.2	155.7	
A_c	1.3	1.3	
Length	1.576	1.576	
Width	0.825	0.825	
N_s	72	72	
I_sc_ref	5.17	5.31	
V_oc_ref	43.99	44.06	
I_mp_ref	4.78	4.9	
V_mp_ref	36.63	36.72	
alpha_sc	0.002146	0.002204	
beta_oc	-0.159068	-0.159321	
T_NOCT	49.9	49.9	
a_ref	1.9817	1.98841	
I_L_ref	5.1757	5.31615	

(continues on next page)

(continued from previous page)

```
I_o_ref                          1.14916e-09                      1.22524e-09
R_s                                0.316688                         0.299919
R_sh_ref                            287.102                          259.048
Adjust                              16.0571                           16.419
gamma_r                             -0.5072                          -0.5072
BIPV                                      N                                N
Version                   SAM 2018.11.11 r2                SAM 2018.11.11 r2
Date                                1/3/2019                         1/3/2019

           A10Green_Technology_A10J_S72_185 A10Green_Technology_A10J_M60_220  \
Technology                       Mono-c-Si                        Multi-c-Si
Bifacial                                 0                                 0
STC                                184.702                           219.876
PTC                                  160.2                             189.1
A_c                                    1.3                             1.624
Length                               1.576                             1.632
Width                                0.825                             0.995
N_s                                     72                                60
I_sc_ref                              5.43                              7.95
V_oc_ref                             44.14                             36.06
I_mp_ref                              5.03                               7.3
V_mp_ref                             36.72                             30.12
alpha_sc                          0.002253                          0.004357
beta_oc                           -0.15961                         -0.130681
T_NOCT                                49.9                              50.2
a_ref                              1.98482                           1.67309
I_L_ref                            5.43568                           7.95906
I_o_ref                         1.16164e-09                       3.34415e-09
R_s                                0.311962                          0.140393
R_sh_ref                            298.424                           123.168
Adjust                              15.6882                           21.8752
gamma_r                             -0.5072                          -0.5196
BIPV                                      N                                N
Version                   SAM 2018.11.11 r2                SAM 2018.11.11 r2
Date                                1/3/2019                         1/3/2019

           A10Green_Technology_A10J_M60_225 A10Green_Technology_A10J_M60_230  \
Technology                       Multi-c-Si                       Multi-c-Si
Bifacial                                 0                                0
STC                                224.986                          230.129
PTC                                  193.5                            204.1
A_c                                  1.624                            1.624
Length                               1.632                            1.632
Width                                0.995                            0.995
N_s                                     60                               60
I_sc_ref                              8.04                              8.1
V_oc_ref                             36.24                            36.42
I_mp_ref                              7.44                             7.58
V_mp_ref                             30.24                            30.36
alpha_sc                          0.004406                         0.007857
beta_oc                           -0.131334                        -0.130748
T_NOCT                                50.2                             46.4
```

(continues on next page)

(continued from previous page)

```
a_ref                                1.67178                          1.68048
I_L_ref                              8.04721                          8.10361
I_o_ref                          3.01424e-09                      3.09549e-09
R_s                                  0.14737                         0.152058
R_sh_ref                             164.419                          340.983
Adjust                               20.6984                          21.5544
gamma_r                              -0.5196                           -0.493
BIPV                                       N                                N
Version                      SAM 2018.11.11 r2                SAM 2018.11.11 r2
Date                                 1/3/2019                         1/3/2019

           A10Green_Technology_A10J_M60_235 A10Green_Technology_A10J_M60_240  \
Technology                        Multi-c-Si                       Multi-c-Si
Bifacial                                   0                                0
STC                                  235.008                          240.538
PTC                                    208.7                            213.3
A_c                                    1.624                            1.624
Length                                 1.632                            1.632
Width                                  0.995                            0.995
N_s                                       60                               60
I_sc_ref                                8.23                             8.32
V_oc_ref                               36.72                            36.84
I_mp_ref                                7.68                             7.83
V_mp_ref                                30.6                            30.72
alpha_sc                            0.007983                          0.00807
beta_oc                            -0.131825                        -0.132256
T_NOCT                                 46.4                             46.4
a_ref                                1.69698                          1.69423
I_L_ref                              8.23464                          8.32177
I_o_ref                          3.24284e-09                      2.97878e-09
R_s                                 0.151504                         0.150077
R_sh_ref                             268.701                          706.27
Adjust                               21.8719                          20.881
gamma_r                               -0.493                           -0.493
BIPV                                       N                                N
Version                      SAM 2018.11.11 r2                SAM 2018.11.11 r2
Date                                 1/3/2019                         1/3/2019

           A2Peak_Power_POWER_ON_P220_6x10 Aavid_Solar_ASMS_165P  ...  \
Technology                       Multi-c-Si            Multi-c-Si  ...
Bifacial                                  0                     0  ...
STC                                 219.978                164.85  ...
PTC                                     195                 146.3  ...
A_c                                   1.633                 1.301  ...
Length                                1.633                 1.575  ...
Width                                     1                 0.826  ...
N_s                                      60                    72  ...
I_sc_ref                               7.98                  5.25  ...
V_oc_ref                              36.72                  43.5  ...
I_mp_ref                               7.26                  4.71  ...
V_mp_ref                               30.3                    35  ...
alpha_sc                            0.00399              0.001575  ...
```

(continues on next page)

(continued from previous page)

```
beta_oc                          -0.12852             -0.170955  ...
T_NOCT                               47.9                    45  ...
a_ref                             1.59703                 1.96463  ...
I_L_ref                           8.00023                 5.27415  ...
I_o_ref                       7.85133e-10             1.19571e-09  ...
R_s                              0.229644                0.595855  ...
R_sh_ref                          90.5774                 129.523  ...
Adjust                            12.2172                 7.16388  ...
gamma_r                             -0.46                  -0.519  ...
BIPV                                    N                       N  ...
Version                  SAM 2018.11.11 r2       SAM 2018.11.11 r2  ...
Date                             1/3/2019                1/3/2019  ...

           Zytech_Solar_ZT275P Zytech_Solar_ZT280P Zytech_Solar_ZT285P  \
Technology          Multi-c-Si          Multi-c-Si          Multi-c-Si
Bifacial                     0                   0                   0
STC                    275.014             280.329             285.326
PTC                        248               252.6               257.3
A_c                      1.931               1.931               1.931
Length                    1.95                1.95                1.95
Width                     0.99                0.99                0.99
N_s                         72                  72                  72
I_sc_ref                  8.31                 8.4                8.48
V_oc_ref                  45.1               45.25               45.43
I_mp_ref                  7.76                7.87                7.97
V_mp_ref                 35.44               35.62                35.8
alpha_sc              0.004014            0.004057            0.004096
beta_oc              -0.144275           -0.144755           -0.145331
T_NOCT                    46.4                46.4                46.4
a_ref                  1.81027             1.81485              1.8201
I_L_ref                8.32377             8.41015              8.4867
I_o_ref            1.24062e-10          1.2341e-10         1.21696e-10
R_s                   0.566493            0.552584            0.543536
R_sh_ref                341.758             457.468             687.561
Adjust                  5.42178             5.27464             5.06509
gamma_r                 -0.4308             -0.4308             -0.4308
BIPV                          N                   N                   N
Version      SAM 2018.11.11 r2   SAM 2018.11.11 r2   SAM 2018.11.11 r2
Date                  1/3/2019            1/3/2019            1/3/2019

           Zytech_Solar_ZT290P Zytech_Solar_ZT295P Zytech_Solar_ZT300P  \
Technology          Multi-c-Si          Multi-c-Si          Multi-c-Si
Bifacial                     0                   0                   0
STC                    290.036             295.066             300.003
PTC                      261.9               266.5               271.2
A_c                      1.931               1.931               1.931
Length                    1.95                1.95                1.95
Width                     0.99                0.99                0.99
N_s                         72                  72                  72
I_sc_ref                  8.55                8.64                8.71
V_oc_ref                 45.59               45.75               45.96
I_mp_ref                  8.07                8.16                8.26
```

(continues on next page)

(continued from previous page)

```
V_mp_ref                  35.94              36.16              36.32
alpha_sc                0.00413           0.004173           0.004207
beta_oc               -0.145842          -0.146354          -0.147026
T_NOCT                     46.4               46.4               46.4
a_ref                   1.82278            1.83125            1.84441
I_L_ref                 8.55196            8.64154            8.80531
I_o_ref           1.17172e-10        1.21851e-10        1.31413e-10
R_s                    0.538499           0.521134           0.515735
R_sh_ref                2348.68            2917.76             552.455
Adjust                  4.66051             4.9013             5.41555
gamma_r                 -0.4308            -0.4308            -0.4308
BIPV                          N                  N                  N
Version      SAM 2018.11.11 r2  SAM 2018.11.11 r2  SAM 2018.11.11 r2
Date                   1/3/2019           1/3/2019           1/3/2019

           Zytech_Solar_ZT305P Zytech_Solar_ZT310P Zytech_Solar_ZT315P  \
Technology           Multi-c-Si          Multi-c-Si          Multi-c-Si
Bifacial                      0                   0                   0
STC                     305.056             310.144             315.094
PTC                       275.8               280.5               285.1
A_c                       1.931               1.931               1.931
Length                     1.95                1.95                1.95
Width                      0.99                0.99                0.99
N_s                          72                  72                  72
I_sc_ref                   8.87                 8.9                9.01
V_oc_ref                  46.12               46.28               46.44
I_mp_ref                   8.36                8.46                8.56
V_mp_ref                  36.49               36.66               36.81
alpha_sc               0.004284            0.004299            0.004352
beta_oc               -0.147538            -0.14805           -0.148562
T_NOCT                     46.4                46.4                46.4
a_ref                   1.84915              1.8574             1.86502
I_L_ref                 8.87402              8.9948             9.10661
I_o_ref           1.30106e-10        1.34888e-10        1.38664e-10
R_s                    0.506611           0.495904           0.488376
R_sh_ref                1119.07             767.958             682.292
Adjust                  5.24155             5.44634             5.57874
gamma_r                 -0.4308            -0.4308            -0.4308
BIPV                          N                  N                  N
Version      SAM 2018.11.11 r2  SAM 2018.11.11 r2  SAM 2018.11.11 r2
Date                   1/3/2019           1/3/2019           1/3/2019

           Zytech_Solar_ZT320P
Technology           Multi-c-Si
Bifacial                      0
STC                      320.42
PTC                       289.8
A_c                       1.931
Length                     1.95
Width                      0.99
N_s                          72
I_sc_ref                   9.12
```

(continues on next page)

(continued from previous page)

```
V_oc_ref                  46.6
I_mp_ref                  8.66
V_mp_ref                    37
alpha_sc              0.004405
beta_oc              -0.149073
T_NOCT                    46.4
a_ref                  1.87378
I_L_ref                9.21845
I_o_ref           1.44659e-10
R_s                   0.475581
R_sh_ref               604.221
Adjust                 5.83833
gamma_r                -0.4308
BIPV                         N
Version      SAM 2018.11.11 r2
Date                  1/3/2019

[25 rows x 21535 columns]
```

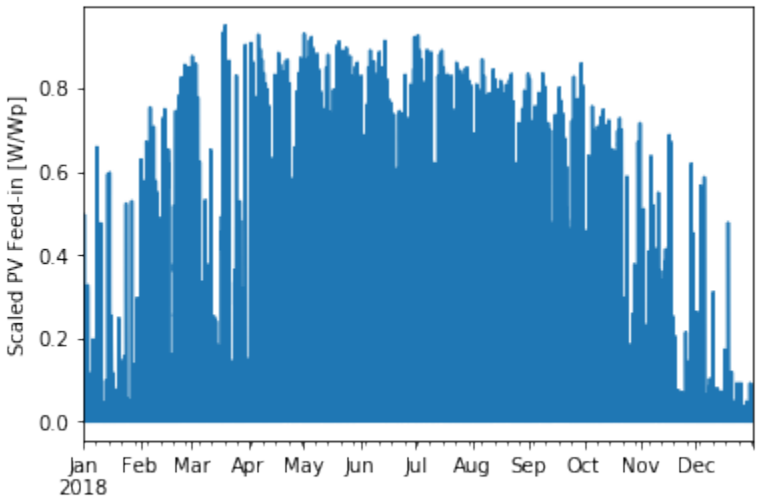
In this example modules and inverter from the manufacturer Canadian Solar are applied. The modules (CS6X-P) use multi-c-Si technololgy and have a nominal power of 300 Wp (STC) and an efficiency of 15.6%. 200 modules are connected to a central inverter that has a nominal AC power of 60 kW.

[10]:
```
pv_system = ar.PVSystem(module='Canadian_Solar_Inc__CS6X_300P',
                        inverter='Canadian_Solar_Inc___CSI_60KTL_CT__480V_')
```

To get the PV plant's electrical power output, we need to provide the irradiance data determined above and the specifications for the orientation of the modules (tilt and azimuth). An azimuth value of 180 represents a surface facing south, a tilt of 0 implies a horizontal module alignment. Furthermore, the electrical output is scaled to its peak power [W/Wp].

[11]:
```
pv_feed_in = pv_system.get_feedin(
    weather=df_solar, location=solar.location,
    surface_tilt=30, surface_azimuth=180,  # South
    modules_per_string=20, strings_per_inverter=10, scaling='peak_power')  # [W/Wp]
```

[12]:
```
# Plot the PV feed-in (scaled to peak power)
pv_feed_in.plot()
plt.ylabel('Scaled PV Feed-in [W/Wp]')
plt.show()
```



2.3.3 SolarThermalCollector class

The solar-thermal collector requires data for the plane of array irradiance (POA). The POA combines the direct normal (DNI) irradiance with sky diffuse and ground-reflected irradiance components and is returned as an OrderedDict or DataFrame via method `get_plane_of_array_irradiance`. The method needs information about the array's orientation (tilt and azimuth) as input arguments. Additional keyword arguments (e.g., "albedo") can be specified via kwargs.

[13]:
```
poa = solar.get_plane_of_array_irradiance(
    surface_tilt=45, surface_azimuth=180)
```

[14]:
```
# Print the components of POA data
poa.iloc[7:17]
```

[14]:
```
                           poa_global  poa_direct  poa_diffuse  \
2018-01-01 07:00:00+00:00    0.000000   -0.000000     0.000000
2018-01-01 08:00:00+00:00   41.916672   22.033527    19.883144
2018-01-01 09:00:00+00:00  186.436000  133.391016    53.044984
2018-01-01 10:00:00+00:00  315.071214  240.922697    74.148516
2018-01-01 11:00:00+00:00  523.571366  454.400115    69.171251
2018-01-01 12:00:00+00:00  380.848212  291.253417    89.594795
2018-01-01 13:00:00+00:00   76.660323   22.057950    54.602373
2018-01-01 14:00:00+00:00  119.564815   79.493429    40.071386
2018-01-01 15:00:00+00:00   17.410464    0.000000    17.410464
2018-01-01 16:00:00+00:00    0.000000   -0.000000     0.000000

                           poa_sky_diffuse  poa_ground_diffuse
2018-01-01 07:00:00+00:00         0.000000            0.000000
2018-01-01 08:00:00+00:00        18.967853            0.915291
2018-01-01 09:00:00+00:00        49.790614            3.254369
2018-01-01 10:00:00+00:00        68.758468            5.390049
2018-01-01 11:00:00+00:00        61.645523            7.525729
2018-01-01 12:00:00+00:00        82.984357            6.610437
2018-01-01 13:00:00+00:00        52.161596            2.440777
```

(continues on next page)

(continued from previous page)

```
2018-01-01 14:00:00+00:00        37.935706            2.135680
2018-01-01 15:00:00+00:00        16.596871            0.813592
2018-01-01 16:00:00+00:00         0.000000            0.000000
```

Set up a solar collector array (type: evacuated tube collectors). The internal procedure to calculate the collector thermal energy output is adopted from reference: *V.Quaschning, 'Regenerative Energiesysteme', 10th edition, Hanser, 2019, p.131ff.* Characteristic values for the parametrization of different collector types are also indicated in this reference.

[15]:
```
solar_collector_data = ar.SolarThermalCollector(
    optical_efficiency=0.80, thermal_loss_parameter_1=1.1,
    thermal_loss_parameter_2=0.008, irradiance_data=poa['poa_global'],
    t_ambient=data['T_amb [C]'], t_collector_in=60, t_collector_out=90)
```

The thermal energy output of the solar collector is calculated and returned by method `get_collector_heat_output`.

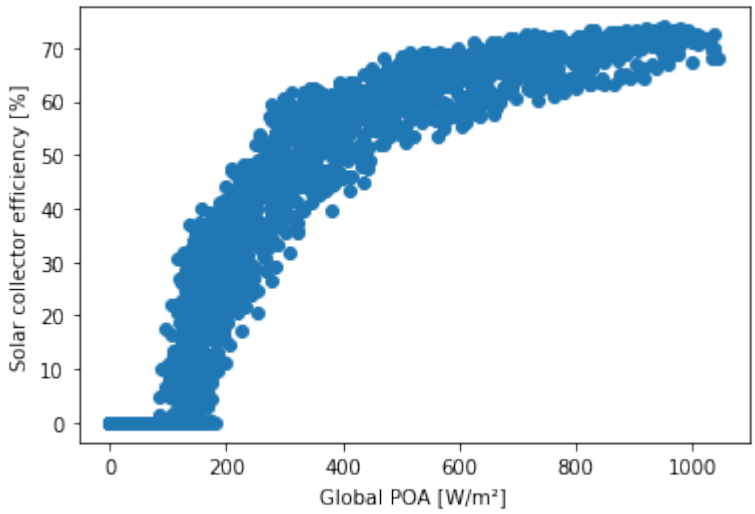
[16]:
```
solar_heat = solar_collector_data.get_collector_heat_output()
```

[17]:
```
solar_heat[7:17]
```

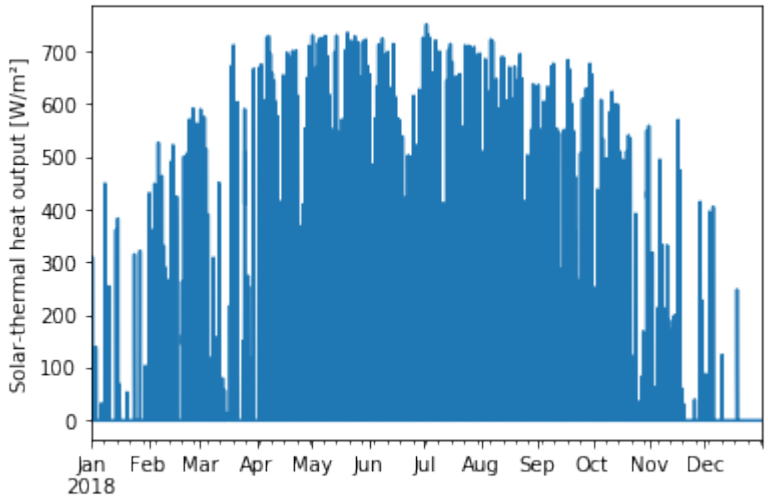
[17]:
```
2018-01-01 07:00:00+00:00      0.000000
2018-01-01 08:00:00+00:00      0.000000
2018-01-01 09:00:00+00:00     38.884480
2018-01-01 10:00:00+00:00    142.444971
2018-01-01 11:00:00+00:00    310.112613
2018-01-01 12:00:00+00:00    197.446090
2018-01-01 13:00:00+00:00      0.000000
2018-01-01 14:00:00+00:00      0.000000
2018-01-01 15:00:00+00:00      0.000000
2018-01-01 16:00:00+00:00      0.000000
dtype: float64
```

The data section above shows that despite present irradiation on the collector in the hours 1 pm to 3 pm, no useful thermal energy is released. That is explainable by the high share of heat losses at large temperature differences and a low total irradiance. The following figure shows that depending on the outside temperature, the collector efficiency can even drop to zero. Thus, no further thermal energy is extracted.

[18]:
```
plt.scatter(poa['poa_global'], solar_collector_data.collector_efficiency*100)
plt.xlabel('Global POA [W/m²]'), plt.ylabel('Solar collector efficiency [%]')
plt.show()
```



[19]:
```
# Plot the specific solar-thermal heat output
solar_heat.plot()
plt.ylabel('Solar-thermal heat output [W/m²]')
plt.show()
```



2.3.4 Simple solar model

The following simple model exemplarily shows how to use the generated feed-in time series of the PVSystem and the SolarThermalCollector classes in an *aristopy* model. The model could easily be extended, e.g., by adding a heat pump or a (seasonal) thermal storage system.

[20]:
```
# Create energy system instance
es = ar.EnergySystem()
```

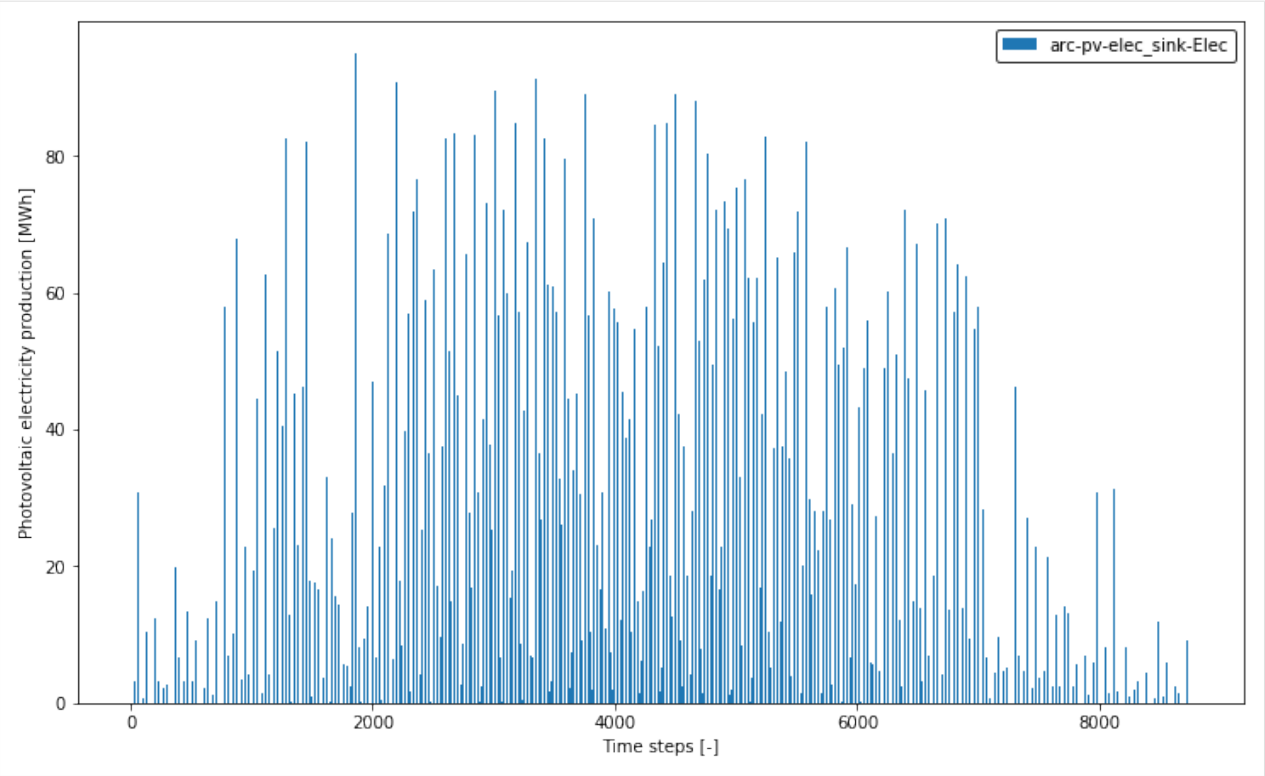
The feed-in time series of the PV system can be introduced directly to the model with an electricity providing instance of the Source class. Since the applied time series is scaled to the peak power value, we need to multiply this relative value with the overall capacity of the PV plant in a user expression.

[21]:
```
# Add a photovoltaic component (electricity source)
pv = ar.Source(
    ensys=es, name='pv', outlet=ar.Flow('Elec', 'elec_sink'),
    time_series_data=ar.Series('pv_feed_in', pv_feed_in),  # [MW/MWp]
    capacity=100, capex_per_capacity=700e3, opex_per_capacity=0.025*700e3,  # [MWp]
    user_expressions='Elec == CAP * pv_feed_in')
```

The Source class is also used to impose the solar-thermal heat output. The available time-series provides area-specific data, so the delivered thermal energy is calculated as the product of the time series values and the collector area. The area is introduced as an additional variable without time dependency via the "additional_vars" keyword. Since also the costs of the plant are area-related [EUR/m^2], the Area varibale is defined as the basic variable of the component (see *Component class API*).

[22]:
```
# Add a Solar-thermal collector component (heat source)
solar_collector = ar.Source(
    ensys=es, name='solar_collector', outlet=ar.Flow('Heat', 'heat_sink'),
    basic_variable='Area', additional_vars=ar.Var('Area', has_time_set=False),
    time_series_data=ar.Series('collector_feedin', solar_heat / 1e6),  # [MW/m^2]
    user_expressions='Heat == collector_feedin * Area',
    capacity=1e5, capex_per_capacity=300, opex_per_capacity=300*0.01)  # [m^2], [EUR/m^2]

# Add sinks for electricity and heat
elec_sink = ar.Sink(ensys=es, name='elec_sink', inlet=ar.Flow('Elec'))
heat_sink = ar.Sink(ensys=es, name='heat_sink', inlet=ar.Flow('Heat'))

# Run the optimization
es.optimize(tee=False)
```

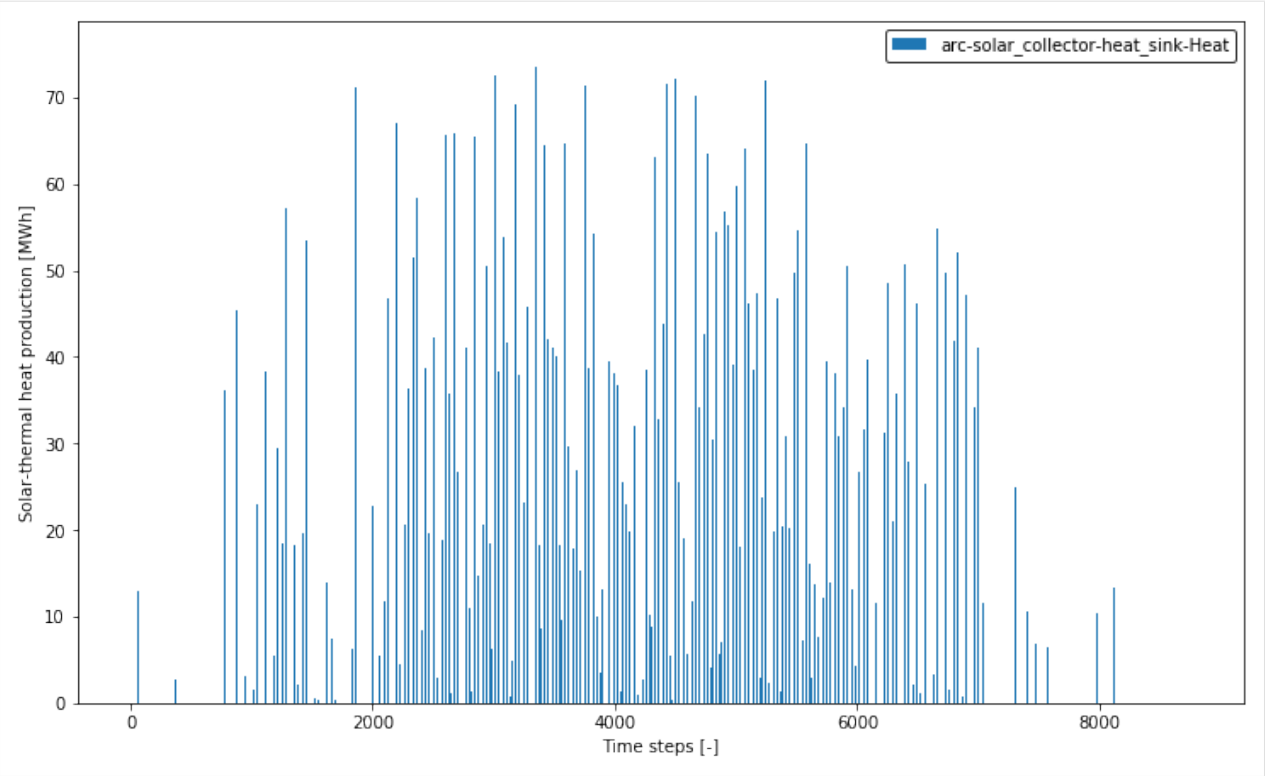
[23]:
```
# Plot PV feed-in to electricity sink
plotter = ar.Plotter(json_file='results.json')
plotter.plot_operation('elec_sink', 'Elec', show_plot=True,
                       ylabel='Photovoltaic electricity production [MWh]')
```



[24]:
```
print(f'Capacity PV system:    {pv.block.CAP.value} MW')
print(f'Electricity PV system: {sum(pv.block.Elec.get_values().values())} MWh/a')
```

```
Capacity PV system:    100.0 MW
Electricity PV system: 133830.5074627531 MWh/a
```

[25]:
```
# Plot solar collector feed-in to heat sink
plotter.plot_operation('heat_sink', 'Heat', show_plot=True,
                       ylabel='Solar-thermal heat production [MWh]')
```



[26]:
```
print(f'Area solar collector:  {solar_collector.block.Area.value} m²')
print(f'Heat solar collector:  {sum(solar_collector.block.Heat.get_values().values())}␣
↪MWh/a')
```

```
Area solar collector:  100000.0 m²
Heat solar collector:  78976.54196102788 MWh/a
```

CHAPTER

THREE

PACKAGE DESCRIPTION

The two main elements of *aristopy* are the EnergySystem and the added Component instances. These components are modeled with the help of the superordinate class Component and the inheriting subclasses Source, Sink, Conversion, Bus, and Storage. Moreover *aristopy* comes with some auxiliary classes that are required for the modeling process (Flow, Series, Var), or can be helpful for debugging (Logger), visualizing results (Plotter), or modeling specific solar components (SolarThermalCollector, PVSystem).

Contents:

3.1 EnergySystem

- File name: energySystem.py
- Last edited: 2020-06-14
- Created by: Stefan Bruche (TU Berlin)

The EnergySystem class is aristopy's main model container. An instance of the EnergySystem class holds the modeled components, the overall pyomo model and the results of the optimization. The EnergySystem class provides features to built and solve the optimization problem, manipulate the associated component models, and process the results of the optimization. The implemented class methods are:

- *cluster*: Perform clustering of the implemented time series data
- *declare_model*: Declare the pyomo optimization model
- *optimize*: Call the main optimization routine
- *relax_integrality*: Relax the integrality of binary variables
- *edit_component_variables*: Edit properties of component variables, e.g., change bounds or domains
- *reset_component_variables*: Reset component variables after applying changes, e.g., relaxation
- *export_component_configuration*, *import_component_configuration*,: Export and import configurations, i.e. component existences and capacities
- *add_design_integer_cut_constraint*: Create integer-cut-constraints to exclude the current design solution from the solution space and enforce a new design in subsequent model runs
- *add_variable*, *add_constraint*, *add_objective_function_contribution*: Add variables, constraints and objective function contributions directly to the main pyomo model, outside of the component declaration

class aristopy.energySystem.**EnergySystem**(*number_of_time_steps=8760*, *hours_per_time_step=1*, *interest_rate=0.05*, *economic_lifetime=20*, *logging=None*)

Initialize an instance of the EnergySystem class.

Parameters

- **number_of_time_steps** (*int (>0)*) – Number of considered time steps for modeling the dispatch problem. With "hours_per_time_step" the share of the modeled year can be calculated. In this way, the cost of each time step is scaled and included in the objective function. *Default: 8760*
- **hours_per_time_step** (*int (>0)*) – Number of hours per modeled time step. *Default: 1*
- **interest_rate** (*float, int (>=0)*) – Value to calculate the present value factor of a cost rate that occurs in the future. *Default: 0.05 (corresponds to 5%)*
- **economic_lifetime** (*int (>0)*) – Years to consider for calculating the net present value of an investment with annual incoming and outgoing cash flows. *Default: 20*
- **logging** (*None or instance of aristopy's "Logger" class*) – Specify the behavior of the logging by setting an own Logger class instance. User can decide where to log (file/console) and what to log (see description of aristopy "Logger"). *Default: None (display minimal logging in the console)*

add_variable(*var*)

Function to manually add pyomo variables to the main pyomo model (ConcreteModel: model) of the energy system instance via instances of aristopy's Var class. The attributes of the variables are stored in DataFrame "added_variables" and later initialized during the call of function 'optimize', or 'declare_model'.

Parameters **var** – Instances of aristopy's Var class (single or in list)

add_constraint(*rule*, *name=None*, *has_time_set=True*, *alternative_set=None*)

Function to manually add constraints to the main pyomo model after the instance has been created. The attributes are stored in the DataFrame 'added_constraints' and later initialized during the call of function 'optimize', or 'declare_model'.

Parameters

- **rule** (*function*) – A Python function that specifies the constraint with a equality or inequality expression. The rule must hold at least two arguments: First the energy system instance it is added to (in most cases: self), second the ConcreteModel of the instance (model). Additional arguments represent sets (e.g., time).
- **name** (*str*) – Name (identifier) of the added constraint. The rule name is used if no name is specified. *Default: None*
- **has_time_set** (*bool*) – Is True if the time set of the energy system model is also a set of the added constraint. *Default: True*
- **alternative_set** – Alternative constraint sets can be added here via iterable Python objects (e.g. list). *Default: None*

add_objective_function_contribution(*rule*, *name=None*)

Additional objective function contributions can be added with this method. The method requires a Python function input that takes the main pyomo model (ConcreteModel: model) and returns a single (scalar) value.

Parameters

- **rule** (*function*) – A Python function returning a scalar value which is added to the objective function of the model instance. The rule must hold exactly two arguments: The energy system instance it is added to (in most cases: self), second the ConcreteModel of the instance (model).
- **name** (*str*) – Name (identifier) of the added objective function contribution. The rule name is used if no name is specified. *Default: None*

cluster(*number_of_typical_periods=4*, *number_of_time_steps_per_period=24*, *cluster_method='hierarchical'*, ***kwargs*)

Method for the aggregation and clustering of time series data. First, the time series data and their respective weights are collected from all components and split into pieces with equal length of 'number_of_time_steps_per_period'. Subsequently, a clustering method is called and each period is assigned to one of 'number_of_typical_periods' typical periods. The clustered data is later stored in the components. The package tsam (time series aggregation module) is used to perform the clustering. The clustering algorithm can be controlled by adding required keyword arguments (using 'kwargs' parameter). To learn more about tsam and possible keyword arguments see the package documentation.

Parameters

- **number_of_typical_periods** (*int (>0)*) – Number of typical periods to be clustered. *Default: 4*
- **number_of_time_steps_per_period** (*int (>0)*) – Number of time steps per period *Default: 24*
- **cluster_method** (*str*) – Name of the applied clustering method (e.g., 'k_means'). See the tsam documentation for all possible options. *Default: 'hierarchical'*

declare_time_sets(*model*, *use_clustered_data*)

Initialize time parameters and four different time sets.

The "time_set" represents the general set. The index holds tuples of periods and time steps inside of these periods. In case the optimization is performed without time series aggregation, the set runs from [(0,0), (0,1), ..to.., (0,number_of_time_steps-1)]. Otherwise: [(0,0), …, (0,number_of_time_steps_per_period-1), (1,0), …, (number_of_typical_periods-1, number_of_time_steps_per_period-1)]. The set "intra_period_time_set" holds tuples of periods and points in time before, after or between regular time steps inside of a period. Hence, the second value runs from 0 to "number_of_time_steps" (without aggregation) or "number_of_time_steps_per_period" respectively. The third set "inter_period_time_set" is one-dimensional and ranges from 0 to the overall number of periods plus 1 (1 + number_of_time_steps / number_of_time_steps_per_period). It is empty if no aggregation is used. The "typical_periods_set" is a set ranging from 0 to the number of typical periods. If no aggregation is used it only holds 0.

E.g.: Case with 2 periods and 3 time steps per period 1) _____(0,0)___(0,1)___(0,2)_____(1,0)___(1,1)___(1,2)____ 2) __(0,0)___(0,1)___(0,2)___(0,3)___(1,0)___(1,1)___(1,2)___(1,3)_ 3) 0_____ 1 _____2 4) _____ 0 _____ 1 _____ 1) time_set, 2) intra_period_time_steps_set, 3) inter_period_time_steps_set, 4) typical_periods_set

Parameters

- **model** (*pyomo ConcreteModel*) – Pyomo model instance holding sets, variables, constraints and the objective function.
- **use_clustered_data** (*bool*) – Declare optimization model with original (full scale) time series (=> False) or clustered data (=> True).

declare_objective(*model*)

Method to declare the objective function of the optimization problem. The objective function contributions (CAPEX and OPEX) are calculated in the components itself, and collected and summarized by the 'declare_objective' function of the EnergySystem instance. The objective function of the optimization is the maximization of the net present value.

Parameters **model** (*pyomo ConcreteModel*) – Pyomo model instance holding sets, variables, constraints and the objective function.

declare_model(*use_clustered_data=False*, *declare_persistent=False*, *persistent_solver='gurobi_persistent'*)

Declare the pyomo optimization model of the EnergySystem instance. First, all component connections are

established by using their input and output specifications and the plausibility of connections is validated. Then, the ConcreteModel instance is created and time sets, component model blocks, variables, ports, constraints, arcs, and the objective function are added.

Parameters

- **use_clustered_data** (*bool*) – Declare optimization model with original (full scale) time series (=> False) or clustered data (=> True). *Default: False*
- **declare_persistent** (*bool*) – States if a persistent model instance should be formed. In this case, after model declaration a persistent solver instance is created and the declared model instance is assigned. *Default: False*
- **persistent_solver** (*str*) – Name of the persistent solver to be used. Possible options are "gurobi_persistent" and "cplex_persistent". Is ignored if keyword "declare_persistent" is False. *Default: 'gurobi_persistent'*

optimize(*declare_model=True*, *use_clustered_data=False*, *declare_persistent=False*, *persistent_solver='gurobi_persistent'*, *solver='gurobi'*, *time_limit=None*, *optimization_specs=''*, *results_file='results.json'*, *tee=True*)

Call the optimization routine for the EnergySystem instance. First, a new pyomo model instance is built (if 'declare_model'=True), then the model is delivered to the specified solver and finally, the optimization results are exported to a json-file.

Parameters

- **declare_model** (*bool*) – Declare a new instance of the pyomo ConcreteModel (=> True: Call function 'declare_model') or use a previously declared model instance if available (=> False). *Default: True*
- **use_clustered_data** (*bool*) – Declare optimization model with original (full scale) time series (=> False) or clustered data (=> True). *Default: False*
- **declare_persistent** (*bool*) – States if a persistent model instance should be formed. In this case, after model declaration a persistent solver instance is created and the declared model instance is assigned. *Default: False*
- **persistent_solver** (*str*) – Name of the persistent solver to be used. Possible options are "gurobi_persistent" and "cplex_persistent". Is ignored if keyword "declare_persistent" is False. *Default: 'gurobi_persistent'*
- **solver** (*str*) – Name of the applied solver (make sure the solver is available on your machine and aristopy can find the path to the solver executables). *Default: 'gurobi'*
- **time_limit** (*int (>0) or None*) – Limits the total optimization time (in seconds). If the time limit is reached, the solver returns with the best currently found solution. If None is specified, solver runs until the default time limit is exceeded (solver specific) or another criteria for abortion is triggered (e.g., reached 'MIPGap'). *Default: None*
- **optimization_specs** (*str*) – Additional solver parameters can be set from a string. To find out more about possible solver options check the documentation of the applied solver (e.g., see gurobi documentation) E.g.: 'Threads=1 MIPGap=0.01' *Default: ''*
- **results_file** (*str or None*) – Name of the results file (required format: .json). *Default: 'results.json'*
- **tee** (*bool*) – Show solver output (on screen and/or logfile). *Default: True*

export_component_configuration(*write_to_excel=False*, *file_name='component_config.xlsx'*)

This function collects the configuration data of all modeled components (results of the optimization) as pandas Series and returns them in a pandas DataFrame. The configuration features are (if exist):

- the binary existence variable (utils.BI_EX),
- the binary existence variables of modules (utils.BI_MODULE_EX)
- the component capacity variable (utils.CAP)

Parameters

- **write_to_excel** (*bool*) – Specify whether the component configuration DataFrame should be stored in an Excel-file. *Default: False*
- **file_name** (*str*) – Name of the exported Excel-file containing the component configuration (if flag "write_to_excel" is True). *Default: 'component_config.xlsx'*

Returns The configuration of all components of the model instance.

Return type pandas DataFrame

import_component_configuration(*config*, *fix_existence=True*, *fix_modules=True*, *fix_capacity=True*, *store_previous_variables=True*)

Function to load a pandas DataFrame with configuration specifications (binary existence variables and capacity variable values) and fix the configuration of the modeled components accordingly.

Parameters

- **config** (*pandas DataFrame*) – The component configuration of the model instance (generated by function 'export_component_configuration').
- **fix_existence** (*bool*) – Specify whether the imported (global) binary component existence variables should be fixed (if available). *Default: True*
- **fix_modules** (*bool*) – Specify whether the imported binary existence variables of the component modules should be fixed (if available). *Default: True*
- **fix_capacity** (*bool*) – Specify whether the imported component capacity variable should be fixed or not. *Default: True*
- **store_previous_variables** (*bool*) – State whether the representation of the variables before applying the configuration import should be stored in DataFrame "variables_copy" of each component. This representation can be used by function "reset_component_variables" to undo changes. *Default: True*

add_design_integer_cut_constraint(*which_instances='all'*, *include_existence=True*, *include_modules=True*)

Function to add an integer cut constraint to the Concrete Model of the EnergySystem instance. Hereby, the currently calculated design solution is excluded from the solution space. Hence, in a next optimization run of the same model instance a different design solution has to be determined. The integer cuts are destroyed once a new optimization model is declared.

Parameters

- **which_instances** (*str 'all' or list of component (group) names*) – State which components should be considered while formulating the integer cut constraint. The argument can either take the string 'all' or a list of component (group) names. *Default: 'all'*
- **include_existence** (*bool*) – State if the binary existence variables of the components should be considered (if available) for formulating the integer cut constraint. *Default: True*
- **include_modules** (*bool*) – State if the binary modules existence variables of the components should be considered (if available) for formulating the integer cut constraint. *Default: True*

relax_integrality(*which_instances='all'*, *include_existence=True*, *include_modules=True*, *include_time_dependent=True*, *store_previous_variables=True*)

Function to relax the integrality of the binary variables of the modelled components. This means binary variables are declared to be 'NonNegativeReals with an upper bound of 1. The relaxation can be performed for the binary existence variable, the module existence binary variables and time-dependent binary variables.

Parameters

- **which_instances** (*str 'all' or list of component (group) names*) – State for which components the relaxation of the binary variables should be done. The keyword argument can either take the string 'all' or a list of component (group) names. *Default: 'all'*
- **include_existence** (*bool*) – State whether the integrality of the binary existence variables should be relaxed (if available). *Default: True*
- **include_modules** (*bool*) – State whether the integrality of the binary modules existence variables should be relaxed (if available). *Default: True*
- **include_time_dependent** (*bool*) – State whether the integrality of the time-dependent binary variables should be relaxed. (if available). *Default: True*
- **store_previous_variables** (*bool*) – State whether the representation of the variables before applying the relaxation should be stored in the DataFrame "variables_copy" of each component. This representation can be used by function "reset_component_variables" to undo changes. *Default: True*

edit_component_variables(*name*, *which_instances='all'*, *store_previous_variables=True*, ***kwargs*)

Method for manipulating the specifications of already defined component variables (e.g., change variable domain, add variable bounds, etc.).

Parameters

- **name** (*str*) – Name / identifier of the edited variable.
- **which_instances** (*str 'all' or list of component (group) names*) – State for which components the relaxation of the binary variables should be done. The keyword argument can either take the string 'all' or a list of component (group) names. *Default: 'all'*
- **store_previous_variables** (*bool*) – State whether the representation of the variables before applying the relaxation should be stored in the DataFrame "variables_copy" of each component. This representation can be used by function "reset_component_variables" to undo changes. *Default: True*
- **kwargs** – Additional keyword arguments for editing. Options are: 'ub' and 'lb' to add an upper or lower variable bound, 'domain' to set the variable domain, and 'has_time_set' to define if the variable should inherit the global time set of the EnergySystem.

reset_component_variables(*which_instances='all'*)

Function to reset the variables of the modeled components to their state that is stored in the component DataFrame "variables_copy". This includes the resetting of the DataFrame "variables" and the pyomo variables itself (if constructed).

Parameters **which_instances** (*str 'all' or list of component (group) names*) – State for which components the variable resetting should be done. The keyword argument can either take the string 'all' or a list of component (group) names. *Default: 'all'*

serialize()

This method collects all relevant input data and optimization results from the EnergySystem instance, the

added components and the pyomo model instance. The data is arranged in an ordered dictionary, that is later exported as a file in JSON-format.

Returns OrderedDict

3.2 Component

- File name: component.py
- Last edited: 2020-06-30
- Created by: Stefan Bruche (TU Berlin)

Components are added to an instance of the EnergySystem class. The Component class holds basic parameters (e.g., DataFrames for variables and parameters) and methods (e.g., calculate contributions to the objective function), that are required in all types of components. *aristopy* has five basic component types that are inheriting the parameters and methods from the Component class: Source, Sink, Conversion, Bus, Storage. The Component class itself contains abstract methods and can, therefore, not be instantiated itself.

3.2.1 Source and Sink

- File name: sourceSink.py
- Last edited: 2020-06-14
- Created by: Stefan Bruche (TU Berlin)

Sources and sinks are responsible for the transportation of commodities across the system boundary into and out of the EnergySystem instance. The Sink class inherits from the Source class. Both have the same input parameters with only one exception: The Sink has an "inlet" instead of an "outlet" attribute.

Source

`class aristopy.sourceSink.Source(`*ensys*, *name*, *outlet*, *basic_variable='outlet_variable'*, *has_existence_binary_var=False*, *has_operation_binary_var=False*, *time_series_data=None*, *scalar_params=None*, *additional_vars=None*, *user_expressions=None*, *capacity=None*, *capacity_min=None*, *capacity_max=None*, *capex_per_capacity=0*, *capex_if_exist=0*, *opex_per_capacity=0*, *opex_if_exist=0*, *opex_operation=0*, *commodity_rate_min=None*, *commodity_rate_max=None*, *commodity_rate_fix=None*, *commodity_cost=0*, *commodity_revenues=0*, ***kwargs*)

Initialize an instance of the Source class.

Note: See the documentation of the *Component* class for a description of all keyword arguments and inherited methods.

Parameters

- **commodity_rate_min** (*int, or float, or aristopy Series, or None*) – Scalar value or time series that provides a minimal value (lower bound) for the basic variable (typically, Sink inlet commodity, or Source outlet commodity) for every time step. *Default: None*

- **commodity_rate_max** (*int, or float, or aristopy Series, or None*) – Scalar value or time series that provides a maximal value (upper bound) for the basic variable (typically, Sink inlet commodity, or Source outlet commodity) for every time step. *Default: None*
- **commodity_rate_fix** (*int, or float, or aristopy Series, or None*) – Scalar value or time series that provides a fixed value for the basic variable (typically, Sink inlet commodity, or Source outlet commodity) for every time step. *Default: None*
- **commodity_cost** (*int, or float, or aristopy Series*) – Incurred costs for the use / expenditure of the basic variable. Keyword argument takes scalar values or time series data (Note: scalar values provide the same functionality like keyword argument 'opex_operation'). *Default: 0*
- **commodity_revenues** (*int, or float, or aristopy Series*) – Accruing revenues associated with the allocation of the basic variable. Keyword argument takes scalar values or time series data. *Default: 0*

declare_component_constraints(*ensys*, *model*)

Method to declare all component constraints.

The following constraint methods are inherited from the Component class and are not documented in this sub-class:

- *con_couple_bi_ex_and_cap*
- *con_cap_min*
- *con_bi_var_ex_and_op_relation*
- *con_couple_op_binary_and_basic_var*

Method is not intended for public access!

Parameters

- **ensys** – Instance of the EnergySystem class
- **model** – Pyomo ConcreteModel of the EnergySystem instance

con_operation_limit(*model*)

The basic variable of a component is limited by its nominal capacity. This usually means, the operation (main commodity) of a sink / source (MWh) is limited by its nominal power (MW) multiplied with the number of hours per time step. E.g.: `Q[p, t] <= CAP * dt`

Method is not intended for public access!

con_commodity_rate_min(*model*)

The basic variable of a component needs to have a minimal value of "commodity_rate_min" in every time step. (Without correction with "hours_per_time_step" because it should already be included in the time series). E.g.: `Q[p, t] >= op_rate_min[p, t]`

Method is not intended for public access!

con_commodity_rate_max(*model*)

The basic variable of a component can have a maximal value of "commodity_rate_max" in every time step. (Without correction with "hours_per_time_step" because it should already be included in the time series). E.g.: `Q[p, t] <= op_rate_max[p, t]`

Method is not intended for public access!

con_commodity_rate_fix(*model*)
The basic variable of a component needs to have a value of "commodity_rate_fix" in every time step. (Without correction with "hours_per_time_step" because it should already be included in the time series). E.g.: `Q[p, t] == op_rate_fix[p, t]`

Method is not intended for public access!

get_objective_function_contribution(*ensys*, *model*)
Calculate the objective function contributions of the component and add the values to the component dictionary "comp_obj_dict".

Method is not intended for public access!

Parameters

- **ensys** – Instance of the EnergySystem class
- **model** – Pyomo ConcreteModel of the EnergySystem instance

serialize()
This method collects all relevant input data and optimization results from the Component instance, and returns them in an ordered dictionary.

Returns OrderedDict

Sink

class aristopy.**Sink**(*ensys*, *name*, *inlet*, *basic_variable='inlet_variable'*, *has_existence_binary_var=False*, *has_operation_binary_var=False*, *time_series_data=None*, *scalar_params=None*, *additional_vars=None*, *user_expressions=None*, *capacity=None*, *capacity_min=None*, *capacity_max=None*, *capex_per_capacity=0*, *capex_if_exist=0*, *opex_per_capacity=0*, *opex_if_exist=0*, *opex_operation=0*, *commodity_rate_min=None*, *commodity_rate_max=None*, *commodity_rate_fix=None*, *commodity_cost=0*, *commodity_revenues=0*)
Initialize an instance of the Sink class.

The Sink class inherits from the Source class. Both have the same input parameters with only one exception: The Sink has an "inlet" instead of an "outlet" attribute.

Note: See the documentation of the *Component* class and the *Source* class for a description of all keyword arguments and inherited methods.

3.2.2 Conversion

- File name: conversion.py
- Last edited: 2020-06-14
- Created by: Stefan Bruche (TU Berlin)

A conversion component takes commodities at the inlet and provides other commodities at the outlet after an internal conversion.

class aristopy.conversion.**Conversion**(*ensys*, *name*, *inlet*, *outlet*, *basic_variable*, *has_existence_binary_var=False*, *has_operation_binary_var=False*, *time_series_data=None*, *scalar_params=None*, *additional_vars=None*, *user_expressions=None*, *capacity=None*, *capacity_min=None*, *capacity_max=None*, *capacity_per_module=None*, *maximal_module_number=None*, *capex_per_capacity=0*, *capex_if_exist=0*, *opex_per_capacity=0*, *opex_if_exist=0*, *opex_operation=0*, *start_up_cost=0*, *min_load_rel=None*, *instances_in_group=1*, *group_has_existence_order=True*, *group_has_operation_order=True*, *use_inter_period_formulation=True*)

Initialize an instance of the Conversion class.

Note: See the documentation of the *Component* class for a description of all keyword arguments and inherited methods.

Parameters

- **start_up_cost** (*float or int (>=0)*) – Costs incurred when the state of the binary operation variable (BI_OP) changes from 0 (OFF) to 1 (ON) from one time step to the next one [€/Start]; (requires keyword argument 'has_operation_binary_var' set to True). *Default: 0*
- **min_load_rel** (*float or int (0<=value<=1), or None*) – Value for the relative minimal part-load of a conversion unit (e.g., 0.5 represents 50% minimal load). Minimal part-loads require the availability of a binary operation variable ('has_operation_binary_var'=True) and currently, fixed capacities need to be specified ('capacity_min'='capacity_max'). *Default: None*
- **instances_in_group** (*int (>=0)*) – States the number of similar component instances that are simultaneously created and arranged in a group. That means, the user has the possibility to instantiate multiple component instances (only for Conversion!) with identical specifications. These components work independently, but may have an order for their binary existence and/or operation variables (see: 'group_has_existence_order', 'group_has_operation_order'). If a number larger than 1 is provided, the names of the components are extended with integers starting from 1 (e.g., 'conversion_1', ...). *Default: 1*
- **group_has_existence_order** (*bool*) – If multiple similar instances are created and arranged in a group ('instances_in_group'>1), the user might want to introduce an order of their binary existence variable values to break the symmetry of the optimization problem. If the flag is set to True a constraint is created for each component of the group that requires the previous component in the group to exist (BI_EX=1) before the own existence variable can take the value 1. *Default: True*
- **group_has_operation_order** (*bool*) – If multiple similar instances are created and arranged in a group ('instances_in_group'>1), the user might want to introduce an order of their binary operation variable values to break the symmetry of the optimization problem. If the flag is set to True a constraint is created for each component of the group and every time step of the optimization problem that requires the previous component in the group to be operated (BI_OP=1) before the own operation variable can take the value 1. *Default: True*
- **use_inter_period_formulation** (*bool*) – A second binary start-up variable is created (BI_SU_INTER), if the inter-period model formulation is requested (flag=True) and start-up cost are introduced (argument 'start_up_cost'>0). This variable is used in case the model

is optimized with aggregated time-series data (multiple periods). The variable links the binary operation variable values of otherwise independent periods, to enforce start-up cost if the operation status (BI_OP) changes from one period to the next (OFF to ON). Note: This formulation introduces additional variables and constraints, and increases both model accuracy and model complexity. *Default: True*

declare_component_constraints(*ensys*, *model*)
Method to declare all component constraints.

The following constraint methods are inherited from the Component class and are not documented in this sub-class:

- *con_couple_bi_ex_and_cap*
- *con_cap_min*
- *con_cap_modular*
- *con_modular_sym_break*
- *con_couple_existence_and_modular*
- *con_bi_var_ex_and_op_relation*
- *con_couple_op_binary_and_basic_var*

Method is not intended for public access!

Parameters

- **ensys** – Instance of the EnergySystem class
- **model** – Pyomo ConcreteModel of the EnergySystem instance

con_existence_sym_break(*ensys*)
Constraint to state, that the next component in a group can only be built if the previous one already exists (symmetry break constraint for groups consisting of multiple components 'instances_in_group' > 1), if keyword argument 'group_has_existence_order' is set to True. E.g.: `BI_EX (of conversion_2) <= BI_EX (of conversion_1)`

Method is not intended for public access!

con_operation_sym_break(*ensys*, *model*)
Constraint to state, that the next component in a group can only be operated if the previous one is already 'ON' (symmetry break constraint for groups consisting of multiple components 'instances_in_group' > 1), if keyword argument 'group_has_operation_order' is set to True. E.g.: `BI_OP[p, t] (of conversion_2) <= BI_OP[p, t] (of conversion_1)`

Method is not intended for public access!

con_operation_limit(*model*)
The basic variable of a component is limited by its nominal capacity. This usually means, the operation (main commodity) of a conversion (MWh) is limited by its nominal power (MW) multiplied with the number of hours per time step. E.g.: `Q[p, t] <= CAP * dt`

Method is not intended for public access!

con_min_load_rel(*model*)
Constraint to set a value for the relative minimal part-load of a conversion unit (e.g., a component can either be switched off (BI_OP=0) or can work between 50% and 100% of its nominal capacity if a value of 0.5 for 'min_load_rel' is specified. The constraint requires the availability of a binary operation variable ('has_operation_binary_var' is True) and currently, fixed capacities need to be spec-

ified ('capacity_min'='capacity_max'='capacity'). E.g.: `Q[p, t] >= capacity * BI_OP[p, t] * min_load_rel * dt`

Method is not intended for public access!

con_start_up_cost(*model*)
Constraint to determine the status of the binary start-up variable. If the operational status of a component changes from OFF (BI_OP=0) to ON (BI_OP=1) from one time step to the next, the binary start-up variable must take a value of 1. For shut-down and remaining ON / OFF, the status variable can take both values, but the obj. function forces it to be 0. E.g.: `0 <= BI_OP[t-1] - BI_OP[t] + BI_SU[t]`

Method is not intended for public access!

con_start_up_cost_inter(*ensys*, *model*)
Constraint to link the binary operation variables of consecutive periods, to enforce start-up cost if the operation status (BI_OP) changes from one period to the next (OFF to ON). With this, the binary operation status of the last time step in the previous typical period and the operation status of the first time step in the current typical period are compared. Binary variable 'BI_SU_INTER' indicates a potential status change from OFF to ON. E.g.: `0 <= BI_OP[p_typ-1, t_last] - BI_OP[p_typ, 0] + BI_SU_INTER[p]`

Method is not intended for public access!

get_objective_function_contribution(*ensys*, *model*)
Calculate the objective function contributions of the component and add the values to the component dictionary "comp_obj_dict".

Method is not intended for public access!

Parameters

- **ensys** – Instance of the EnergySystem class
- **model** – Pyomo ConcreteModel of the EnergySystem instance

serialize()
This method collects all relevant input data and optimization results from the Component instance, and returns them in an ordered dictionary.

Returns OrderedDict

3.2.3 Bus

- File name: bus.py
- Last edited: 2020-06-14
- Created by: Stefan Bruche (TU Berlin)

A Bus component collects and transfers a commodity. Bus components can also be used to model transmission lines between different sites.

class aristopy.bus.**Bus**(*ensys*, *name*, *inlet*, *outlet*, *basic_variable='inlet_variable'*, *has_existence_binary_var=False*, *time_series_data=None*, *scalar_params=None*, *additional_vars=None*, *user_expressions=None*, *capacity=None*, *capacity_min=None*, *capacity_max=None*, *capex_per_capacity=0*, *capex_if_exist=0*, *opex_per_capacity=0*, *opex_if_exist=0*, *opex_operation=0*, *losses=0*)
Initialize an instance of the Bus class.

Note: See the documentation of the *Component* class for a description of all keyword arguments and inherited methods.

Parameters **losses** (*float or int (0<=value<=1)*) – Factor to specify the relative loss of the transported commodity between inlet and outlet per hour of operation (0 => no loss; 1 => 100% loss). *Default: 0*

declare_component_constraints(*ensys*, *model*)
Method to declare all component constraints.

The following constraint methods are inherited from the Component class and are not documented in this sub-class:

- *con_couple_bi_ex_and_cap*
- *con_cap_min*

Method is not intended for public access!

Parameters

- **ensys** – Instance of the EnergySystem class
- **model** – Pyomo ConcreteModel of the EnergySystem instance

con_operation_limit(*model*)
The basic variable of a component is limited by its nominal capacity. This means, the operation (commodity at inlet or outlet) of a bus (MWh) is limited by its nominal power (MW) multiplied with the number of hours per time step. E.g.: `Q_IN[p, t] <= CAP * dt`

Method is not intended for public access!

con_bus_balance(*model*)
The quantity of the commodity at the outlet must equal the quantity at the inlet minus the the transmission loss share. A bus component cannot store a commodity (correction with "hours_per_time_step" not needed). E.g.: `Q_OUT[p, t] == Q_IN[p, t] * (1 - losses)`

Method is not intended for public access!

serialize()
This method collects all relevant input data and optimization results from the Component instance, and returns them in an ordered dictionary.

Returns OrderedDict

3.2.4 Storage

- File name: storage.py
- Last edited: 2020-06-14
- Created by: Stefan Bruche (TU Berlin)

A storage component can collect a commodity at the inlet at one time step and make it available at the outlet at another time step. Thus, it is a component to provide flexibility.

class aristopy.storage.**Storage**(*ensys*, *name*, *inlet*, *outlet*, *basic_variable='inlet_variable'*, *has_existence_binary_var=False*, *time_series_data=None*, *scalar_params=None*, *additional_vars=None*, *user_expressions=None*, *capacity=None*, *capacity_min=None*, *capacity_max=None*, *capacity_per_module=None*, *maximal_module_number=None*, *capex_per_capacity=0*, *capex_if_exist=0*, *opex_per_capacity=0*, *opex_if_exist=0*, *opex_operation=0*, *charge_rate=1*, *discharge_rate=1*, *self_discharge=0*, *charge_efficiency=1*, *discharge_efficiency=1*, *soc_min=0*, *soc_max=1*, *soc_initial=None*, *use_inter_period_formulation=True*, *precise_inter_period_modeling=False*)

Initialize an instance of the Storage class.

Note: See the documentation of the *Component* class for a description of all keyword arguments and inherited methods.

Parameters

- **charge_rate** (*float or int (>=0)*) – Ratio between the maximum charging power or flow and the storage capacity. It indicates the reciprocal value of the time for a full storage charging process from empty to full (e.g., 'charge_rate'=1/6 => 6 hours needed to load an empty storage fully). *Default: 1*
- **discharge_rate** (*float or int (>=0)*) – Ratio between the maximum discharging power or flow and the storage capacity. It indicates the reciprocal value of the time for a full storage discharging process from full to empty (e.g., 'discharge_rate'=1/6 => 6 hours needed for emptying a fully loaded storage). *Default: 1*
- **self_discharge** (*float or int (0<=value<=1)*) – Share of the storage content that is dissipated and can not be used (e.g., heat losses to the environment for a heat storage). The value is specified in "percent per hour" [%/h]. *Default: 0*
- **charge_efficiency** (*float or int (0<=value<=1)*) – Efficiency value for the charging process. It indicates the ratio between stored and entering commodities. E.g., 'charge_efficiency'=0.9 => for 1 MWh entering the storage in one time step 0.9 MWh are stored, and 0.1 MWh are lost. *Default: 1*
- **discharge_efficiency** (*float or int (0<=value<=1)*) – Efficiency of the discharging process. It indicates the ratio between the usable commodity at the outlet and the reduction in the stored commodity. E.g., 'discharge_efficiency'=0.9 => if SOC is reduced by 1 MWh in one time step, 0.9 MWh are available at the outlet, and 0.1 MWh are lost. *Default: 1*
- **soc_min** (*float or int (0<=value<=1)*) – Relative value to provide a lower bound for the usable storage capacity. E.g., with a storage capacity of 5 MWh and a value for 'soc_min' given with 0.2, the SOC cannot fall below 1 MWh. *Default: 0*
- **soc_max** (*float or int (0<=value<=1)*) – Relative value to provide an upper bound for the usable storage capacity. E.g., with a storage capacity of 5 MWh and a value for 'soc_max' given with 0.8, the SOC cannot exceed 4 MWh. *Default: 1*
- **soc_initial** (*float or int (0<=value<=1), or None*) – Provides a value for the relative state of charge in the first time step of the optimization problem (e.g., 0.5 => 50%). The initial SOC value is applied to all periods if the model has multiple periods (calculation with clustered data), and the keyword argument 'use_inter_period_formulation' is set to False. *Default: None*

- **use_inter_period_formulation** (*bool*) – States whether a model formulation should be applied that connects the states of charge of a storage component for otherwise independent periods (only used if time series aggregation is applied). Additional variables and constraints are created if the keyword argument is set to True. This formulation enables the (energy) transport between periods (especially relevant for long-term storages) and likewise increases the model complexity. *Default: True*
- **precise_inter_period_modeling** (*bool*) – States whether the inter-period formulation should be implemented in a simplified (False) or precise (True) way. The type of formulation influences how the constraints are modeled that enforce the SOC's bounds. If the storage has only a low self-discharge value, it is recommended to choose the simplified version (False). This version introduces some additional variables, but requires a significantly smaller number of constraints (Ref: DOI 10.1016/j.apenergy.2018.01.023). *Default: False*

declare_component_constraints(*ensys*, *model*)
Method to declare all component constraints.

The following constraint methods are inherited from the Component class and are not documented in this sub-class:

- *con_couple_bi_ex_and_cap*
- *con_cap_min*
- *con_cap_modular*
- *con_modular_sym_break*
- *con_couple_existence_and_modular*

Method is not intended for public access!

Parameters

- **ensys** – Instance of the EnergySystem class
- **model** – Pyomo ConcreteModel of the EnergySystem instance

con_operation_limit(*model*)
The state of charge (SOC) of a storage component is limit by its nominal capacity in all time steps. E.g.: `SOC[p, t] <= CAP`

Method is not intended for public access!

con_soc_balance(*ensys*, *model*)
Constraint that connects the state of charge (SOC) of each time step with the charging and discharging events. The change in the SOC between two points in time has to match the values of charging and discharging and the self-discharge of the storage (explicit Euler formulation). Note: The SOC is not necessarily a value between 0 and 1 here. E.g., `SOC[p, t+1] <= SOC[p, t]*(1-self_dischar)**dt + Q_IN[p, t]*eta_char - Q_OUT[p, t]/eta_dischar`

Method is not intended for public access!

con_charge_rate(*ensys*, *model*)
Constraint to limit the value of the charge variable by applying a maximal charge rate. E.g.: `Q_IN[p, t] <= CAP * charge_rate * dt`

Method is not intended for public access!

con_discharge_rate(*ensys*, *model*)
Constraint to limit the value of the discharge variable by applying a maximal discharge rate. E.g.: `Q_OUT[p, t] <= CAP * discharge_rate * dt`

Method is not intended for public access!

con_cyclic_condition(*ensys*, *model*)
Constraint to enforce that the SOC in the last time step of a period (after charging and discharging events) equals the SOC at the beginning of the same period. In case the inter-period formulation is activated, this constraint demands that the cycle condition is also fulfilled for the full time scale problem => SOC in global first time step (e.g., SOC[t=1]) equals SOC in global last time step (e.g., SOC[t=8760]).

Method is not intended for public access!

con_soc_initial(*ensys*, *model*)
Constraint that sets a value for the relative state of charge in the first time step of the optimization problem. The same initial SOC value is applied to all periods if the model has multiple periods. Otherwise, the initial value is only specified for very beginning of the time set. E.g.: `SOC[p, 0] == CAP * soc_initial` or `SOC_INTER[0] == CAP * soc_initial`

Method is not intended for public access!

con_soc_intra_period_start(*ensys*, *model*)
The state of charge consists of two parts (Intra and Inter), if the intra-period model formulation ('use_inter_period_formulation'=True) is selected. This constraint sets the intra-period part (SOC) to zero at the beginning of each period. E.g.: `SOC[p, 0] == 0`

Method is not intended for public access!

con_soc_inter_period_balance(*ensys*, *model*)
Constraint to calculate the inter-period state of charge (SOC_INTER) of the next period from the previous one and the value of the SOC (Intra) in the last time step of the related typical period. The constraint also accounts for self-discharge losses associated with the variable SOC_INTER (=> this can result in steps in the overall SOC profile at the boundary of 2 periods). E.g.: `SOC_INTER[p+1] == SOC_INTER[p]*(1-self_dischar)**( time_steps_per_period*dt) + SOC[typ_p, last_t]`

Method is not intended for public access!

con_soc_bounds_without_inter_period_formulation(*ensys*, *model*)
Two constraints that specify bounds for the state of charge variable (minimal and maximal values) in case the inter-period formulation is NOT selected, or the data is NOT clustered! E.g.: `SOC[p, t] >= CAP * soc_min` and `SOC[p, t] <= CAP * soc_max`

Method is not intended for public access!

con_soc_bounds_with_inter_period_formulation_simple(*ensys*, *model*)
Four constraints that define the bounds for the state of charge variable (minimal and maximal values) in a simplified way in case the inter- period formulation is requested and the data is clustered. The error is relatively small in comparison to the precise method if the specified value for the 'self_discharge' is not too high. However, this version requires a reasonable smaller number of constraints but also some additional variables. See Eq. B1 and B2 in the Appendix of Ref: DOI 10.1016/j.apenergy.2018.01.023.

Method is not intended for public access!

con_soc_bounds_with_inter_period_formulation_precise(*ensys*, *model*)
Two constraints define the bounds for the state of charge variable (minimal and maximal values) in a precise way if the inter-period formulation is requested, and the data is clustered. This version requires two constraints for each time step of the full- scale problem and can be computationally expensive. Users might want to consider applying the simplified constraint formulation (keyword argument 'precise_inter_period_modeling'=False). See Eq. 20 in Ref: DOI 10.1016/j.apenergy.2018.01.023.

Method is not intended for public access!

serialize()

This method collects all relevant input data and optimization results from the Component instance, and returns them in an ordered dictionary.

Returns OrderedDict

class aristopy.component.**Component**(*ensys*, *name*, *inlet*, *outlet*, *basic_variable*, *has_existence_binary_var=False*, *has_operation_binary_var=False*, *time_series_data=None*, *scalar_params=None*, *additional_vars=None*, *user_expressions=None*, *capacity=None*, *capacity_min=None*, *capacity_max=None*, *capacity_per_module=None*, *maximal_module_number=None*, *capex_per_capacity=0*, *capex_if_exist=0*, *opex_per_capacity=0*, *opex_if_exist=0*, *opex_operation=0*)

Initialize an instance of the Component class. Note that an instance of the class Component itself can not be instantiated since it holds abstract methods. The abstract methods are later overwritten by the child classes to enable instantiation.

Parameters

- **ensys** (*instance of aristopy's EnergySystem class*) – EnergySystem instance the component is added to.
- **name** (*str*) – Unique name of the component or the component group.
- **inlet** (*(list of) instance(s) of aristopy's Flow class, or None*) – The inlet holds a set of aristopy's Flow instances, transporting commodities, entering the component. The commodities are creating variables inside of the component. The variable name is consistent with the name of the commodity, if not specified differently in the Flow instance. Example: inlet=aristopy.Flow('ABC', var_name='XYZ') => commodity 'ABC' is entering the component, and is represented by a newly an internally added variable with the name 'XYZ'.
- **outlet** (*(list of) instance(s) of aristopy's Flow class, or None*) – See description of keyword argument 'inlet'.
- **basic_variable** (*str*) – Components may have multiple variables, but every component has only one basic variable. It is used to restrict capacities, set operation rates, and calculate CAPEX and OPEX (if available). Usually, the basic variable points to a commodity variable on the inlet or the outlet of the component. In this case, users need to use string inputs 'inlet_variable' or 'outlet_variable', respectively. If any other variable should be used as the basic variable (e.g., added via keyword argument 'additional_vars'), users need to set the variable name directly, as specified during declaration.
- **has_existence_binary_var** (*bool*) – States if the component has a binary variable that indicates its existence status. If the parameter is set to True, a scalar pyomo binary variable is added to the pyomo block model of the component (name specified in utils-file, default: 'BI_EX'). It can be used to enable minimal component capacities ('capacity_min'), or capacity-independent CAPEX and OPEX ('capex_if_exist', 'opex_if_exist'). *Default: False*
- **has_operation_binary_var** (*bool*) – States if the component has a binary variable that indicates its operational status. If the parameter is set to True, a pyomo binary variable, utilizing the global EnergySystem time set, is added to the pyomo block model of the component (name specified in utils-file, default: 'BI_OP'). It can be used to enable load-depending conversion rates (via 'user_expressions'), or minimal part-loads ('min_load_rel'), and start-up cost ('start_up_cost') (the last two only for Conversion). *Default: False*
- **time_series_data** (*(list of) instance(s) of aristopy's Series class, or None*) – Keyword argument for adding time series data to the component by using in-

stances of aristopy's Series class. This data can be used for manual scripting in the 'user_expressions'. *Default: None*

- **scalar_params** (*dict, or None*) – Keyword argument for adding scalar parameters to the component by using a Python dict {'parameter_name': value}. The scalar parameters can be used for manual scripting in the 'user_expressions'. *Default: None*
- **additional_vars** (*(list of) instance(s) of aristopy's Var class, or None*) – Keyword argument for adding variables to the component, that are not automatically created, e.g., by attaching Flows to the inlet or outlet. The variables are provided by adding a set of aristopy's Var instances. Example: additional_vars=aristopy.Var('ABC', domain='Reals', ub=42) *Default: None*
- **user_expressions** (*(list of) str, or None*) – Keyword argument for adding expression-like strings, which are converted into pyomo constraints during model declaration. User expressions can be applied for various ends, e.g., for specifying commodity conversion rates, or limiting capacities. The options for mathematical operators are: 'sum', 'sin', 'cos', 'exp', 'log', '==', '>=', '<=', '**', '*', '/', '+', '-', '(', ')'. The expressions can handle variables that are created with standard names (see Globals in utils-file, e.g., CAP), and the variables and parameters added by the user via keyword arguments: 'inlet', 'outlet', 'time_series_data', 'scalar_params', 'additional_vars'. Example: user_expressions=['Q == 0.5*F + 2*BI_OP', 'CAP <= 42'] *Default: None*
- **capacity** (*float or int (>=0), or None*) – States the fixed capacity value of the component, if the component exists. Hence, if the parameter 'has_existence_binary_var' is set to True, the value for 'capacity' is only applied if the component is selected in the optimal design (BI_EX=1). *Default: None*
- **capacity_min** (*float or int (>=0), or None*) – States the minimal component capacity, if the component exists. Hence, if the parameter 'has_existence_binary_var' is set to True, the value for 'capacity_min' is only applied if the component is selected in the optimal design (BI_EX=1). *Default: None*
- **capacity_max** (*float or int (>=0), or None*) – States the maximal component capacity. This value introduces an upper bound for the capacity variable. It also serves as an over-estimator and is required if the parameters 'has_existence_binary_var' or 'has_operation_binary_var' is True. *Default: None*
- **capacity_per_module** (*float or int (>=0), or None*) – If a component is modular, its capacity can be modeled with parameter 'capacity_per_module'. If a value is introduced, an additional binary variable is created (default name: 'BI_MODULE_EX'), which indicates the existence of each module. *Default: None*
- **maximal_module_number** (*int (>0), or None*) – This keyword argument works in combination with parameter 'capacity_per_module' and indicates the maximal number of modules to be installed in the optimal design. *Default: None*
- **capex_per_capacity** (*float or int (>=0)*) – Parameter to calculate the capital investment cost associated with the *CAPACITY* of a component. The final value for capacity-related CAPEX is obtained by multiplying 'capex_per_capacity' with the capacity variable value (CAP). *Default: 0*
- **capex_if_exist** (*float or int (>=0)*) – Parameter to calculate the capital investment cost associated with the *EXISTENCE* of a component. The final value for existence-related CAPEX is obtained by multiplying 'capex_if_exist' with the binary existence variable value (BI_EX). *Default: 0*
- **opex_per_capacity** (*float or int (>=0)*) – Parameter to calculate the annual operational cost associated with the *CAPACITY* of a component. The final value for capacity-

related OPEX is obtained by multiplying 'opex_per_capacity' with the capacity variable value (CAP). *Default: 0*

- **opex_if_exist** (*float or int (>=0)*) – Parameter to calculate the annual operational cost associated with the *EXISTENCE* of a component. The final value for existence-related OPEX is obtained by multiplying 'opex_if_exist' with the binary existence variable value (BI_EX). *Default: 0*
- **opex_operation** (*float or int (>=0)*) – Parameter to calculate the annual operational cost associated with the *OPERATION* of a component (associated with its basic variable). The final value for operation-related OPEX is obtained by summarizing the products of 'opex_operation' with the time-dependent values of the basic variable. *Default: 0*

pprint()
Access the pretty print functionality of the pyomo model Block

add_to_energy_system(*ensys*, *group*, *instances_in_group=1*)
Add the component to the specified instance of the EnergySystem class. This implies, creation of copies with unique names if multiple identical components should be instantiated ('instances_in_group' > 1), adding the component to the dictionary 'components' of the EnergySystem, and initializing a Logger instance for each component.

Method is not intended for public access!

Parameters

- **ensys** – EnergySystem instance the component is added to.
- **group** – Name (str) of the group where the component is added to (corresponds to initialization keyword attribute 'name').
- **instances_in_group** – (int > 0) States the number of similar component instances that are simultaneously created and arranged in a group. That means, the user has the possibility to instantiate multiple component instances (only for Conversion!) with identical specifications. These components work independently, but may have an order for their binary existence and/or operation variables (see: 'group_has_existence_order', 'group_has_operation_order'). If a number larger than 1 is provided, the names of the components are extended with integers starting from 1 (e.g., 'conversion_1', ...). *Default: 1*

add_var(*name*, *domain='NonNegativeReals'*, *has_time_set=True*, *alternative_set=None*, *ub=None*, *lb=None*, *init=None*)
Method for adding a variable to the pandas DataFrame 'variables'.

Method is not intended for public access!

Parameters

- **name** – (str) Name / identifier of the added variable.
- **domain** – (str) A super-set of the values the variable can take on. Possible values are: 'Reals', 'NonNegativeReals', 'Binary'. *Default: 'NonNegativeReals'*
- **has_time_set** – (bool) Is True, if the time set of the EnergySystem instance is also a set of the added variable. * Default: True
- **alternative_set** – Alternative variable sets can be added here via iterable Python objects (e.g. list). *Default: None*
- **ub** – (float, int, None) Upper variable bound. *Default: None*
- **lb** – (float, int, None) Lower variable bound. *Default: None*

- **init** – A function or Python object that provides starting values for the added variable. *Default: None*

store_var_copy(*name*, *var*)

Method to store the current specifications of a provided variable in the pandas DataFrame "variables_copy".

Method is not intended for public access!

Parameters

- **name** – (str) Name / identifier of the stored variable.
- **var** – (pandas Series) Variable data to store.

relax_integrality(*include_existence=True*, *include_modules=True*, *include_time_dependent=True*, *store_previous_variables=True*)

Method to relax the integrality of the binary variables. This means binary variables are declared to be 'NonNegativeReals' with an upper bound of 1. This method encompasses the resetting of the DataFrame "variables" and the pyomo variables itself (if already constructed). The relaxation can be performed for the binary existence variable, the module existence binary variables and time-dependent binary variables.

Parameters

- **include_existence** (*bool*) – State whether the integrality of the binary existence variables should be relaxed (if available). *Default: True*
- **include_modules** (*bool*) – State whether the integrality of the binary modules existence variables should be relaxed (if available). *Default: True*
- **include_time_dependent** (*bool*) – State whether the integrality of the time-dependent binary variables should be relaxed. (if available). *Default: True*
- **store_previous_variables** (*bool*) – State whether the representation of the variables before applying the relaxation should be stored in the DataFrame "variables_copy". This representation can be used by method "reset_variables" to undo changes. *Default: True*

edit_variable(*name*, *store_previous_variables=True*, ***kwargs*)

Method on component level for manipulating the specifications of already defined component variables (e.g., change variable domain, add variable bounds, etc.).

Parameters

- **name** (*str*) – Name / identifier of the edited variable.
- **store_previous_variables** (*bool*) – State whether the representation of the variables before applying the edit_variable method should be stored in DataFrame "variables_copy" of each component. This representation can be used by mehtod "reset_variables" to undo changes. *Default: True*
- **kwargs** – Additional keyword arguments for editing. Options are: 'ub' and 'lb' to add an upper or lower variable bound, 'domain' to set the variable domain, and 'has_time_set' to define if the variable should inherit the global time set of the EnergySystem.

reset_variables()

Method to reset the variables to the state, that is stored in the DataFrame "variables_copy". This includes the resetting of the DataFrame "variables" and the pyomo variables itself (if already constructed).

export_component_configuration()

This method exports the component configuration data (results of the optimization) as a pandas Series. The features are (if exist):

- the binary existence variable (utils.BI_EX),
- the binary existence variables of modules (utils.BI_MODULE_EX)

- the component capacity variable (of main commodity, utils.CAP)

Returns The configuration of the modelled component instance.

Return type pandas Series

import_component_configuration(*data*, *fix_existence=True*, *fix_modules=True*, *fix_capacity=True*, *store_previous_variables=True*)

Method to load a pandas Series with configuration specifications (binary existence variables and capacity variable values). The values are used to fix a specific component configuration (for example from other model runs).

Parameters

- **data** (*pandas Series*) – The configuration of the modelled component instance.
- **fix_existence** (*bool*) – Specify whether the imported (global) binary component existence variable should be fixed (if available). *Default: True*
- **fix_modules** (*bool*) – Specify whether the imported binary existence variable of the component modules should be fixed (if available). *Default: True*
- **fix_capacity** (*bool*) – Specify whether the imported component capacity variable (of the main commodity) should be fixed or not. *Default: True*
- **store_previous_variables** (*bool*) – State whether the representation of the variables before applying the configuration import should be stored in DataFrame "variables_copy" of each component. This representation can be used by method "reset_variables" to undo changes. *Default: True*

add_param(*name*, *data*, *tsam_weight=1*)

Method for adding a parameter to the pandas DataFrame 'parameters'.

Method is not intended for public access!

Parameters

- **name** – (str) Name / identifier of the added parameter.
- **data** – (float, int, list, dict, numpy array, pandas Series) Data.
- **tsam_weight** – (int or float) Weighting factor to use in case the provided parameter is a series and an aggregation is requested. *Default: 1*

set_time_series_data(*use_clustered_data*)

Sets the time series data in the 'parameters' DataFrame of a component depending on whether a calculation with aggregated time series data should be performed or with the original data (without clustering).

Method is not intended for public access!

Parameters **use_clustered_data** (*bool*) – Use aggregated data (True), or original data (False).

get_time_series_data_for_aggregation()

Collect all time series data and their respective weights and merge them in two dictionaries. The returned dictionaries are used for the time series aggregation in the "cluster" method of the EnergySystem instance.

Method is not intended for public access!

Returns (dict) time series data, (dict) time series weights

set_aggregated_time_series_data(*data*)

Store the aggregated time series data in the 'parameters' DataFrame.

Method is not intended for public access!

Parameters **data** – (pandas DataFrame) DataFrame with the aggregated time series data of all components.

declare_component_model_block(*model*)
Create a pyomo Block and store it in attribute 'block' of the component. The pyomo Block is the container for all pyomo objects of the component.

Method is not intended for public access!

Parameters **model** – Pyomo ConcreteModel of the EnergySystem instance

declare_component_variables(*model*)
Create all pyomo variables that are stored in DataFrame 'variables'.

Method is not intended for public access!

Parameters **model** – Pyomo ConcreteModel of the EnergySystem instance

declare_component_ports()
Create all ports from the dictionaries 'inlet_commod_and_var_names' and 'outlet_commod_and_var_names', and assign port variables.

Note: The ports are currently always created with extensive behaviour!

Method is not intended for public access!

declare_component_user_constraints(*model*)
Create all constraints, that are introduced by manual scripting in the keyword argument 'user_expressions'.

Method is not intended for public access!

Parameters **model** – Pyomo ConcreteModel of the EnergySystem instance

abstract declare_component_constraints(*ensys*, *model*)
Abstract method to create (conventional) component constraints.

con_couple_bi_ex_and_cap()
Constraint to couple the global existence binary variable with the capacity variable. If the component does not exist, the capacity must take a value of 0. Note: The availability of the required 'capacity_max' parameter is checked during initialization. E.g.: `CAP <= BI_EX * capacity_max`

Method is not intended for public access!

con_cap_min()
Constraint to set the minimum capacity of a component (based on its basic variable). If a binary existence variable is declared, the minimal capacity is only enforced if the component exists. E.g.: `CAP >= capacity_min * BI_EX` or `CAP >= capacity_min`

Method is not intended for public access!

con_cap_modular()
Constraint to calculate the nominal capacity of a component from the product of the capacity per module and the number of existing modules. E.g.: `CAP == capacity_per_module * summation(BI_MODULE_EX)`

Method is not intended for public access!

con_modular_sym_break()
Constraint to state, that the next module can only be built if the previous one already exists (symmetry break constraint for components consisting of multiple modules). E.g.: `BI_MODULE_EX[2] <= BI_MODULE_EX[1]`

Method is not intended for public access!

con_couple_existence_and_modular()
Constraint to couple the global existence binary variable with the binary existence status of the first module. All other modules are indirectly coupled via symmetry breaks. E.g.: `BI_EX >= BI_MODULE_EX[1]`

Method is not intended for public access!

con_bi_var_ex_and_op_relation(*model*)
Constraint to set a relationship between the binary variables for existence and operation. A component can only be operated if it exists. E.g.: `BI_OP[p, t] <= BI_EX`

Method is not intended for public access!

abstract con_operation_limit(*model*)
Constraint to limit the operation (value of the basic variable) of a component (MWh) by its nominal power (MW) multiplied with the number of hours per time step (not for Storage because it is already a capacity!). E.g.: `Q[p, t] <= CAP * dt` (Conversion, Sink, Source) `SOC[p, t] <= CAP` (Storage) `Q_IN[p, t] <= CAP * dt` (Bus)

Method is not intended for public access!

con_couple_op_binary_and_basic_var(*model*)
Constraint to couple the binary operation variable (is available), with the basic variable of the component. Therefore, the "capacity_max" parameter serves for overestimation (BigM). E.g.: `Q[p, t] <= capacity_max * BI_OP[p, t] * dt`

Method is not intended for public access!

get_objective_function_contribution(*ensys*, *model*)
Calculate the objective function contributions of the component and add the values to the component dictionary "comp_obj_dict".

Method is not intended for public access!

Parameters

- **ensys** – Instance of the EnergySystem class
- **model** – Pyomo ConcreteModel of the EnergySystem instance

serialize()
This method collects all relevant input data and optimization results from the Component instance, and returns them in an ordered dictionary.

Returns OrderedDict

3.3 Utilities

There are several utility classes that are necessary or optional for the modeling process. The Flow class is required to add component commodities, variables, and interconnections. The Series and Var classes are used to introduce time-series data and additional variables to the model. The Logger class is a helpful tool for the debugging process. Auto-generated plots of the Plotter class can also be used for result validation. The different Solar classes provide functionality to generate feed-in time series data for thermal and electrical solar collectors.

3.3.1 Flow, Series, Var

- File name: flowSeriesVar.py
- Last edited: 2020-06-14
- Created by: Stefan Bruche (TU Berlin)

The instances of the Flow class are required to add commodities and the associated variables to the modeled components and to create the component interconnections. The Series and Var classes are used to introduce time-series data and additional variables to the pyomo model instance.

Flow

class aristopy.flowSeriesVar.**Flow**(*commodity*, *link=None*, *var_name='commodity_name'*, ***kwargs*)

A Flow represents a connection point of the inlet or the outlet of a component. The Flow has a single commodity that is entering or leaving the component. Additionally, information on the variable name of the commodity inside the component can be provided (default: use the name of the commodity) and the linking component (source or destination of the Flow) can be stated.

Parameters

- **commodity** (*str*) – Name (identifier) of the commodity
- **link** (*str, or None*) – Component name of the connected Flow source or destination needs to be stated on one or both sides of the Flow *Default: None*
- **var_name** (*str*) – Variable name of the Flow commodity as used inside the component, e.g. for scripting of user expressions. *Default: 'commodity_name' => i.e. use name of the commodity*

Series

class aristopy.flowSeriesVar.**Series**(*name*, *data*, *weighting_factor=1.0*)

The Series class is used to add time series data to the model. Time series data can be required in the Source class to implement data for keyword arguments 'commodity_rate_min',˝'commodity_rate_max', 'commodity_rate_fix'. Additionally, they might be needed to a add time- dependent commodity cost or revenues, or generally for scripting of user expressions (added via 'time_series_data' argument).

Parameters

- **name** (*str*) – Name (identifier) of the time series data instance. Can be used for scripting of user expressions.
- **data** (*list, dict, numpy array, pandas Series*) – Time series data
- **weighting_factor** – Weighting factor to use for the clustering *Default: 1.0*
- **weighting_factor** – float or int

Var (Variable)

class aristopy.flowSeriesVar.**Var**(*name*, *domain='NonNegativeReals'*, *has_time_set=True*, *alternative_set=None*, *ub=None*, *lb=None*, *init=None*)

Class to manually add pyomo variables to a component (via argument "additional_vars"), or the main model container (ConcreteModel: model) of the EnergySystem instance (via function "add_variable").

Parameters

- **name** (*str*) – Name (identifier) of the added variable
- **domain** (*str*) – A super-set of the values the variable can take on. Possible values are: 'Reals', 'NonNegativeReals', 'Binary'. *Default: 'NonNegativeReals'*
- **has_time_set** (*bool*) – Is True if the time set of the EnergySystem instance is also a set of the added variable. *Default: True*
- **alternative_set** – Alternative variable sets can be added here via iterable Python objects (e.g. list) *Default: None*
- **ub** (*float or int*) – Upper variable bound. *Default: None*
- **lb** (*float or int*) – Lower variable bound. *Default: None*
- **init** – A function or Python object that provides starting values for the added variable. *Default: None*

3.3.2 Logger

- File name: logger.py
- Last edited: 2020-06-14
- Created by: Stefan Bruche (TU Berlin)

An instance of the Logger class can be assigned to the keyword 'logging' during initialization of the EnergySystem instance. That way, the user can decide where to log (file, console, or both), and what to log ('DEBUG', 'ERROR', ...). The Logger output is usually helpful during debugging.

class aristopy.logger.**Logger**(*logfile_name='logfile.log'*, *delete_old_logs=True*, *default_log_handler='stream'*, *local_log_handler={}*, *default_log_level='DEBUG'*, *local_log_level={}*, *write_screen_output_to_logfile=False*)

The Logger class can be used to manage the logging of the EnergySystem instance and the associated component instances.

Parameters

- **logfile_name** (*str*) – Name of the file to store the logging.
- **delete_old_logs** (*bool*) – Delete old log-files before creating a new file? Otherwise new logs might be appended to an old log-file.
- **default_log_handler** (*str*) – Sets a default handler to the loggers. The handler options are "file" or "stream":
  - "file" sends logs to a file with the specified "logfile_name"
  - "stream" sends logs to the console (sys.stdout).
- **local_log_handler** (*dict*) – Dictionary to override the default handler level for specified instance types ('EnergySystem', 'Source', 'Sink', 'Conversion', 'Storage', 'Bus'). E.g.: ... = {'Bus': 'file'}

- **default_log_level** (*str*) – Sets the default threshold for the logger level for all derived instance loggers. Logging messages which are less severe than level will be ignored. Options for logger levels are: 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'.
- **local_log_level** (*dict*) – Dictionary to override the default log level for specified instance types ('EnergySystem', 'Source', 'Sink', 'Conversion', 'Storage', 'Bus'). E.g.: ... = {'Bus': 'WARNING', ...}
- **write_screen_output_to_logfile** (*bool*) – State whether the output that is printed on the console should be written to the logfile too.

get_logger(*instance*)

Generate a new logger instance with default logger level and default logger handle. The name of the logger is specified in "instance". If an instance of class 'EnergySystem', 'Source', 'Sink', 'Conversion', 'Storage', 'Bus' is passed to the function, local log handler and local log levels might be used.

Parameters **instance** – String (name of the logger) or instance of aristopy class type 'EnergySystem', 'Source', 'Sink', 'Conversion', 'Storage', 'Bus'.

Returns New logger instance

3.3.3 Plotter

- File name: plotter.py
- Last edited: 2020-06-14
- Created by: Stefan Bruche (TU Berlin)

The Plotter class provides three basic plotting methods:

- *plot_operation*: A mixed bar and line plot that visualizes the operation of a component on the basis of a selected commodity.
- *plot_objective*: Bar chart that summarizes the cost contributions of each component to the overall objective function value.
- *quick_plot*: Quick visualization for the values of one component variable as a line, scatter, or bar plot.

Note: The results of the optimization are exported to dictionaries and stored as strings in a json-file to easily handle multidimensional indices (e.g. tuples). To evaluate the Python strings we use the function "literal_eval" from the python built in library "ast". (the strings can only consist of: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and None) [Ref]

---

class aristopy.plotter.**Plotter**(*json_file*)

The Plotter class provides three basic plotting methods:

- plot_operation
- plot_objective
- quick_plot

Parameters **json_file** – Path to the optimization results file in JSON-Format

plot_objective(*show_plot=False*, *save_plot=True*, *file_name='objective_plot'*, ***kwargs*)

Method to create a bar chart that summarizes the cost contributions of each component of the EnergySystem instance to the overall objective function value.

Parameters

- **show_plot** (*bool*) – State whether the plot should be shown once finalized *Default: False*
- **save_plot** (*bool*) – State whether the plot should be saved once finalized *Default: True*
- **file_name** (*str*) – Name of the file (if saved); no file-ending required *Default: 'objective_plot'*
- **kwargs** – Additional keyword arguments to manipulate the plot (e.g., labels, figure size, legend position, ...). See dict 'props' of the Plotter class.

quick_plot(*component_name*, *variable_name*, *kind='bar'*, *save_plot=False*, *file_name=None*)

Method to create a quick visualization for the values of one component variable as a line, scatter, or bar plot.

Parameters

- **component_name** (*str*) – Name of the component that holds the variable of interest.
- **variable_name** (*str*) – Name of the variable (or parameter) that should be plotted.
- **kind** (*str*) – States the kind of plot. Possible options are: 'plot' (line plot), 'scatter', 'bar'. *Default: 'bar'*
- **save_plot** (*bool*) – State whether the plot should be saved once finalized *Default: False*
- **file_name** (*str*) – Name of the file (if saved); no file-ending required. Name is autogenerated if None is provided and plot should be saved. *Default: None*

plot_operation(*component_name*, *commodity*, *level_of_detail=2*, *scale_to_hourly_resolution=False*, *plot_single_period_with_index=None*, *show_plot=False*, *save_plot=True*, *file_name='operation_plot'*, ***kwargs*)

Method to create a mixed bar and line plot that visualizes the operation of a component on the basis of a selected commodity.

Parameters

- **component_name** (*str*) – Name of the component that holds the commodity of interest.
- **commodity** (*str*) – Name of the commodity that should be plotted.
- **level_of_detail** (*int (1 or 2)*) – Specifies the level of plotting detail. Only the commodity in the component itself is plotted if 1 is selected. The composition of the commodity (from which sources formed and to which destinations sent) is visualized if 2 is selected. *Default: 2*
- **scale_to_hourly_resolution** (*bool*) – States if the data should be scaled to hourly resolution before plotting. This might be useful, if the optimization was performed with a value for the EnergySystem keyword argument 'hours_per_time_step' larger than 1. *Default: False*
- **plot_single_period_with_index** (*int or None*) – States if only one period with the given index number should be plotted. This is only possible if the optimization was performed with aggregated time series data. *Default: None*
- **show_plot** (*bool*) – State whether the plot should be shown once finalized *Default: False*
- **save_plot** (*bool*) – State whether the plot should be saved once finalized *Default: True*
- **file_name** (*str*) – Name of the file (if saved); no file-ending required *Default: 'operation_plot'*
- **kwargs** – Additional keyword arguments to manipulate the plot (e.g., labels, figure size, legend position, ...). See dict 'props' of the Plotter class.

3.3.4 Solar

- File name: solar.py
- Last edited: 2020-06-30
- Created by: Stefan Bruche (TU Berlin)

The solar classes SolarData, SolarThermalCollector, and PVSystem provide functionality to model feed-in time series data for solar components (thermal and electrical) at a certain location and with specific tilt and azimuth angles.

Note: The solar classes require the availability of the Python module *pvlib*. The module is not provided with the standard installation of *aristopy*. If you want to use the solar classes, consider installing the module in your current environment, e.g. via:

```
>> pip install pvlib
```

For further information and an installation guide, users are referred to the pvlib documentation.

SolarData

class aristopy.solar.**SolarData**(*ghi*, *dhi*, *latitude*, *longitude*, *altitude=0*, *dni=None*)

Class to provide solar input data for PV or solar thermal calculations.

The main output is accessed via function "get_plane_of_array_irradiance" for the POA data with specified surface tilt and azimuth values and "get_irradiance_dataframe" for a pandas DataFrame consisting of GHI, DHI and DNI values at the specified location and time index.

Parameters

- **ghi** – Pandas series (with datetime index and time zone) for global horizontal irradiation data at the respective location.
- **dhi** – Pandas series (with datetime index and time zone) for diffuse horizontal irradiation data at the respective location.
- **latitude** – Latitude value (float, int) for respective location.
- **longitude** – Longitude value (float, int) for respective location.
- **altitude** – Altitude value (float, int) for respective location.
- **dni** – Pandas series (with datetime index and time zone) for direct normal (beam) irradiation data at the respective location. Is calculated from GHI and DHI if not provided here.

property ghi

property data_index

property dhi

property dni

property location

get_dni()

get_plane_of_array_irradiance(*surface_tilt*, *surface_azimuth*, ***kwargs*)

Calculate and return the plane of array irradiance (POA).

Parameters

- **surface_tilt** – tilt of the modules (0=horizontal, 90=vertical)
- **surface_azimuth** – module azimuth angle (180=facing south)
- **kwargs** – Option to specify more keyword arguments, e.g, 'albedo' or 'surface_type'. Search for method 'get_total_irradiance' in the documentation of pvlib for further information.

Returns pandas DataFrame with POA ('poa_global', ...)

get_irradiance_dataframe()
Create and return a pandas DataFrame consisting of global (GHI) and diffuse horizontal (DHI) and direct normal irradiation (DNI).

Returns pandas DataFrame with column names 'ghi', 'dhi', 'dni'

SolarThermalCollector

class aristopy.solar.**SolarThermalCollector**(***kwargs*)
Required input arguments (either while creating the class object or while calling function 'get_collector_heat_output'.

- 'optical_efficiency': Opt. eff of the collector (float, int)
- 'thermal_loss_parameter_1': Th. loss of the collector (float, int)
- 'thermal_loss_parameter_2': Th. loss of the collector (float, int) => See equation in: V.Quaschning, 'Regenerative Energiesysteme', 10th edition, Hanser, 2019, p.131ff.
- 'irradiance_data': Irradiance (POA) on collector array (pd.Series)
- 't_ambient': Ambient temperature (float, int, pd.Series)
- 't_collector_in': Collector inlet temperature (float, int, pd.Series)
- 't_collector_out': Collector outlet temp. (float, int, pd.Series)

property t_collector_in

property t_collector_out

property irradiance_data

property optical_efficiency

property thermal_loss_parameter_1

property thermal_loss_parameter_2

property t_ambient

get_collector_heat_output(***kwargs*)
Required input arguments (either while creating the class object or while calling function 'get_collector_heat_output'.

- 'optical_efficiency': Opt. eff of the collector (float, int)
- 'thermal_loss_parameter_1': Th. loss of the collector (float, int)
- 'thermal_loss_parameter_2': Th. loss of the collector (float, int) => See equation in: V.Quaschning, 'Regenerative Energiesysteme', 10th edition, Hanser, 2019, p.131ff.
- 'irradiance_data': Irradiance (POA) on collector array (pd.Series)
- 't_ambient': Ambient temperature (float, int, pd.Series)

- 't_collector_in': Collector inlet temperature (float, int, pd.Series)
- 't_collector_out': Collector outlet temp. (float, int, pd.Series)

Returns pandas Series of provided collector heat output

PVSystem

class aristopy.solar.**PVSystem**(*module*, *inverter*)

PVSystem class holds a type of PV module and PV inverter.

The main class function is "get_feedin", to get the power feedin for a PV plant with specified module, inverter, tilt and azimuth angle, location and weather data.

Parameters

- **module** – Name of the module as in PVLib Database. To see the full database type e.g. "pvlib.pvsystem.retrieve_sam(name='cecmod')"
- **inverter** – Name of the inverter as in PVLib Database. To see the database type e.g. "pvlib.pvsystem.retrieve_sam(name='cecinverter')"

property module

property module_parameters

property inverter

property inverter_parameters

property mode

property location

property area

Get area of the PV system in m^2

Returns PV System area

property peak_power

PV system peak power [W] can be limited by the inverter or the modules (minimum). If DC mode is selected the inverter is not considered.

Returns Peak power of the PV System

set_location(*latitude*, *longitude*, *altitude*)

get_feedin(*weather*, *surface_tilt*, *surface_azimuth*, *scaling=None*, *mode='ac'*, ***kwargs*)

Parameters

- **weather** – requires pandas DataFrame with at least two out of three column names: 'ghi', 'dhi', 'dni'. Additionally users can specify column names 'wind_speed' and 'temp_air' (used in calc. of losses)
- **surface_tilt** – tilt of the PV modules (0=horizontal, 90=vertical)
- **surface_azimuth** – module azimuth angle (180=facing south)
- **scaling** –

a) None=no feed-in scaling [W],

b) 'area'=scale feed-in to area [W/m2],

c) 'peak_power'=scale feed-in to nominal power [-]

- **mode** –

  a) 'ac': return AC feed-in (including inverter),

  b) 'dc': return DC feed-in (excluding inverter)

- **kwargs** – Examples for kwargs are 'albedo', 'modules_per_string', 'strings_per_inverter', 'temperature_model_parameters', …

Returns pandas DataFrame with POA ('poa_global', …)

CHAPTER

FOUR

LICENSE

MIT License

CHAPTER

FIVE

ACKNOWLEDGMENT

This work was developed during the research project “MINLP-Optimization of Design and Operation of Complex Energy Systems”, funded by the German Federal Ministry for Economic Affairs and Energy (project reference number 03ET4053A). The funding is gratefully acknowledged.

The developers of *aristopy* also want to thank the developer groups of the open-source packages FINE, oemof, tsam, and pvlib. We learned a lot from these packages during the development of *aristopy* and had the opportunity to adapt selected code elements. These packages greatly contribute to the open-source community, and they are worth looking at more closely.

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

A

B

C

D

E

F

G

V