urbs Documentation

Release 1.0.0

tum-ens

Jul 19, 2021

Contents

Author Johannes Dorfner, <johannes.dorfner@tum.de>

Organization Chair of Renewable and Sustainable Energy Systems, Technical University of Munich, <urbs@ens.ei.tum.de>

Version 1.0.0

Date Jul 19, 2021

CHAPTER 1

Contents

1.1 User's manual

These documents give a general overview and help you getting started from after the installation (which is covered in the README.md file on GitHub) to you first running model.

1.1.1 Users guide

Welcome to urbs. The following sections will help you get started.

Overview model structure

urbs is a generator for linear energy system optimization models.

urbs consists of several **model entities**. These are commodities, processes, transmission and storage. Demand and intermittent commodity supply through are modelled through time series datasets.

Commodity

Commodities are goods that can be generated, stored, transmitted and consumed. By convention, they are represented by their energy content (in MWh), but can be changed (to J, kW, t, kg) by simply using different (consistent) units for all input data. Each commodity must be exactly one of the following six types:

- Stock: Buyable at any time for a given price. Supply can be limited per timestep or for a whole year. Examples are coal, gas, uranium or biomass.
- SupIm: Supply intermittent stands for fluctuating resources like solar radiation and wind energy, which are available according to a timeseries of values, which could be derived from weather data.

- Demand: These commodities have a timeseries for the requirement associated and must be provided by output from other process or from storage. Usually, there is only one demand commodity called electricity (abbreviated to Elec), but multiple (e.g. electricity, space heating, process heat, space cooling) demands can be specified.
- Env: The special commodity CO2 is of this type and represents the amount (in tons) of greenhouse gas emissions from processes. Its total amount can be limited, to investigate the effect of policies on the model.
- Buy/Sell: Commodities of these two types can be traded with an external market. Similar to Stock commodities they can be limited per hour or per year. As opposed to Stock commodities the price at which they can be traded is not fixed but follows a user defined time series.

Stock and environmental commodities have three numeric attributes that represent their price, total annual and per timestep supply or emission limit, respectively. Environmental commodities (i.e. CO2) have a maximum allowed quantity that may be created across the entire modeling horizon.

Commodities are defined over the tuple `(year, site, commodity, type)`, for example `(2020, 'Norway', 'Wind', 'SupIm')` for wind in Norway with a time series or `(2020, 'Iceland', 'Electricity', 'Demand')` for an electricity demand time series in Iceland.

Process

Processes describe conversion technologies from one commodity to another. They can be visualised like a black box with input(s) (commodity) and output(s) (commodity). Process input and output ratios are the main technical parameters for processes. Fixed costs for investment and maintenance (per capacity) and variable costs for operation (per output) are the economical parameters.

Processes are defined over two tuples. The first tuple `(year, site, process)` specifies the location of a given process e.g. `(2030, 'Iceland', 'Turbine')` would locate a process `Turbine` at site `Iceland`. The second tuple `(year, process, commodity, direction)` then specifies the inputs and outputs for that process. For example, `(2030, 'Turbine', 'Geothermal', 'In')` and `(2030, 'Turbine', 'Electricity', 'Out')` describes that the process named `Turbine` has a single input `Geothermal` and the single output `Electricity`.

Transmission

Transmission allows instantaneous transportation of commodities between sites. It is characterised by an efficiency and costs, just like processes. Transmission is defined over the tuple `(year, site in, site out, transmission, commodity)`. For example, `(2030, 'Iceland', 'Norway', 'Undersea cable', 'Electricity')` would represent an undersea cable for electricity between Iceland and Norway.

Storage

Storage describes the possibility to deposit a deliberate amount of energy in the form of one commodity at one time step; with the purpose of retrieving it later. Efficiencies for charging/discharging depict losses during input/output. Storage is characterised by capacities both for energy content (in MWh) and charge/discharge power (in MW). Both capacities have independent sets of investment, fixed and variable cost parameters to allow for a very flexible parametrization of various storage technologies; ranging from batteries to hot water tanks.

Storage is defined over the tuple `(year, site, storage, stored commodity)`. For example, `(2020, 'Norway', 'Pump storage', 'Electricity')` represents a pump storage power plant in Norway that can store and retrieve energy in form of electricity.

Time series

Demand

Each combination `(year, site, demand commodity)` may have one time series, describing the aggregate demand (typically MWh) for a commodity within a given timestep. They are a crucial input parameter, as the whole optimization aims to satisfy these demands with minimal costs by the given technologies (process, storage, transmission). An additional feature for demand commodities is demand side management (DSM) which allows for the shifting of demands in time.

Intermittent Supply

Each combination `(year, site, supim commodity)` must be supplied with one time series, normalized to a maximum value of 1 relative to the installed capacity of a process using this commodity as input. For example, a wind power time series should reach value 1, when the wind speed exceeds the modeled wind turbine's design wind speed is exceeded. This implies that any non-linear behaviour of intermittent processes can already be incorporated while preparing this timeseries.

Buy/Sell prices

Each combination `(year, Buy/sell commodity)` must be supplied with one time series which represents the price for purchasing/selling the given commodities in the given modeled year.

Time variable efficiency

Each combination `(year, site, process)` can optionally be supplied with one time series which multiplies the outputs of the process with an acoording factor.

Get started

Welcome to urbs! Here you can learn how to use the program and what to do to create your own optimization problems and run them.

Inputs

There are two different types of inputs the user has to make in order to set up and solve an optimization problem with urbs.

First, there are the model parameters themselves, i.e. the parameters specifying the behavior of the different model entities such as commodities or processes. These parameters are entered into spreadsheets with a standardized structure. These then have to be placed in the subfolder `Input`. There can be no further information given on those parameters here since they make up the particular energy system

models. There are, however, two examples provided with the code, which are explained elsewhere in this documentation.

Second, there are the settings of the modeling run such as the modeling horizon or the solver to be employed. These settings are made in a run script. For the standard example such scripts are given named runme.py for the example `mimo-example` and runBP.py for the example `Business park`. To run a modeling run you then simply execute the according run script by typing:

```
$ python3 runscript.py
```

in the command prompt.

You can immediately test this after the installation by running one of the two standard examples using the corresponding example run scripts.

runscript explained

The runscript can be subdivided into several parts. These will be discussed here in detail.

Imports

The script starts with importing the relevant python libraries as well as the module `urbs`.

```
import os
import shutil
import urbs
```

The included packages have the following functions:

- os and shutil are builtin Python modules, included here for their data path and copying operations.
- urbs is the directory which includes the modules, whose functions are used mainly in this script. These are `prepare_result_directory()`, `setup_solver()` and `run_scenario()`.

More functions can be found in the document API reference.

In the following sections the user defined input, output and scenario settings are described.

Input Settings

The script starts with the specification of the input files, which is either a single .xlsx file located in the same folder as the runscript or a collection of .xlsx files located in the subfolder `Input`:

```
input_files = 'Input'
result_name = 'Mimo-ex'
result_dir = urbs.prepare_result_directory(result_name)  # name + time␣
↪stamp

# copy input file to result directory
try:
    shutil.copytree(input_files, os.path.join(result_dir, 'Input'))
except NotADirectoryError:
```

(continues on next page)

(continued from previous page)

```
    shutil.copyfile(input_files, os.path.join(result_dir, input_files))
# copy runme.py to result directory
shutil.copy(__file__, result_dir)
```

The input file/folder and the runscript are automatically copied into the result folder.

Next variables specifying the desired solver and objective function are set:

```
# choose solver (cplex, glpk, gurobi, ...)
solver = 'glpk'

# objective function
objective = 'cost'  # set either 'cost' or 'CO2' as objective
```

The solver has to be licensed for the specific user, where the open source solver "glpk" is used as the standard. For the objective function urbs currently allows for two options: "cost" and "CO2" (case sensitive). In the former case the total system cost and in the latter case the total CO2-emissions are minimized.

The model parameters are finalized with a specification of timestep length and modeled time horizon:

```
# simulation timesteps
(offset, length) = (3500, 168)  # time step selection
timesteps = range(offset, offset+length+1)
dt = 1  # length of each time step (unit: hours)
```

The variable `timesteps` is the list of timesteps to be simulated. Its members must be a subset of the labels used in `input_file`'s sheets "Demand" and "SupIm". It is one of the function arguments to `create_model()` and accessible directly, so that one can quickly reduce the problem size by reducing the simulation `length`, i.e. the number of timesteps to be optimised. Variable `dt` is the duration of each timestep in the list in hours, where any positiv real value is allowed.

`range()` is used to create a list of consecutive integers. The argument `+1` is needed, because `range(a,b)` only includes integers from `a` to `b-1`:

```
>>> range(1,11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Output Settings

The desired output is also specified by the user in the runscript. It is split into two parts: reporting and plotting. The former is used to generate an excel output file and the latter for standard graphs.

Reporting

urbs by default generates an .xlsx-file as an ouput in result_dir. This file includes all commodities of interest to the user and can be specified as report tuples each consisting of a given `year`, `sites` and `commodities` combination. Information about these commodities is summarized both in sum (in sheet "Energy sums") and as individual timeseries (in sheet "... timeseries").

```
# detailed reporting commodity/sites
report_tuples = [
    (2019, 'North', 'Elec'),
    (2019, 'Mid', 'Elec'),
    (2019, 'South', 'Elec'),
    (2019, ['North', 'Mid', 'South'], 'Elec'),
    (2024, 'North', 'Elec'),
    (2024, 'Mid', 'Elec'),
    (2024, 'South', 'Elec'),
    (2024, ['North', 'Mid', 'South'], 'Elec'),
    (2029, 'North', 'Elec'),
    (2029, 'Mid', 'Elec'),
    (2029, 'South', 'Elec'),
    (2029, ['North', 'Mid', 'South'], 'Elec'),
    (2034, 'North', 'Elec'),
    (2034, 'Mid', 'Elec'),
    (2034, 'South', 'Elec'),
    (2034, ['North', 'Mid', 'South'], 'Elec'),
    ]
```

optional: define names for sites in report_tuples report_sites_name = {('North', 'Mid', 'South'): 'All'}

Optionally it is possible to define clusters of sites for aggregated information and with `report_sites_name` it is then possible to name these. If they are empty, the default value will be taken.

Plotting

urbs generates default result images. Which images exactly are desired can be set by the user. via the following input lines:

```
# plotting commodities/sites
plot_tuples = [
    (2019, 'North', 'Elec'),
    (2019, 'Mid', 'Elec'),
    (2019, 'South', 'Elec'),
    (2019, ['North', 'Mid', 'South'], 'Elec'),
    (2024, 'North', 'Elec'),
    (2024, 'Mid', 'Elec'),
    (2024, 'South', 'Elec'),
    (2024, ['North', 'Mid', 'South'], 'Elec'),
    (2029, 'North', 'Elec'),
    (2029, 'Mid', 'Elec'),
    (2029, 'South', 'Elec'),
    (2029, ['North', 'Mid', 'South'], 'Elec'),
    (2034, 'North', 'Elec'),
    (2034, 'Mid', 'Elec'),
    (2034, 'South', 'Elec'),
    (2034, ['North', 'Mid', 'South'], 'Elec'),
    ]

# optional: define names for sites in plot_tuples
plot_sites_name = {('North', 'Mid', 'South'): 'All'}

# plotting timesteps
```

(continues on next page)

(continued from previous page)

```
plot_periods = {
    'all': timesteps[1:]
    }
```

The logic is similar to the reporting case discussed above. With the setting of plotting timesteps the exact range of the plotted result can be set. In the default case shown this range is all modeled timesteps. For larger optimization timestep ranges this can be impractical and instead the following syntax can be used to hard code which steps are to be plotted exactly.

```
# plotting timesteps
plot_periods = {
    'win': range(1000:1168),
    'sum': range(5000:5168)
    }
```

In this example two 1 week long ranges are plotted between the specified time steps. Using this make sure, that the chosen ranges are subsets of the modeled time steps themselves.

The plot colors can be customized using the module constant `COLORS`. All plot colors are user-definable by adding color `tuple() (r, g, b)` or modifying existing tuples for commodities and plot decoration elements. Here, new colors for displaying import/export are added. Without these, pseudo-random colors are generated in `to_color()`.

```
# create timeseries plot for each demand (site, commodity) timeseries
for sit, com in prob.demand.columns:
```

Scenarios

This section deals with the definition of different scenarios. Starting from the same base scenarios, defined by the data in `input_file`, they serve as a short way of defining the difference in input data. If needed, completely separate input data files could be loaded as well.

The `scenarios` list in the end of the input file allows then to select the scenarios to be actually run.

```
scenarios = [
             urbs.scenario_base,
             urbs.scenario_stock_prices,
             urbs.scenario_co2_limit,
             urbs.scenario_co2_tax_mid,
             urbs.scenario_no_dsm,
             urbs.scenario_north_process_caps,
             urbs.scenario_all_together
            ]
```

The following scenario functions are specified in the subfolder `urbs` in script `scenarios.py`.

Scenario functions

A scenario is simply a function that takes the input `data` and modifies it in a certain way. with the required argument `data`, the input data `dict`.:

```
# SCENARIOS
def scenario_base(data):
    # do nothing
    return data
```

The simplest scenario does not change anything in the original input file. It usually is called "base" scenario for that reason. All other scenarios are defined by 1 or 2 distinct changes in parameter values, relative to this common foundation.:

```
def scenario_stock_prices(data):
    # change stock commodity prices
    co = data['commodity']
    stock_commodities_only = (co.index.get_level_values('Type') == 'Stock')
    co.loc[stock_commodities_only, 'price'] *= 1.5
    return data
```

For example, `scenario_stock_prices()` selects all stock commodities from the `DataFrame` `commodity`, and increases their *price* value by 50%. See also pandas documentation Selection by label for more information about the `.loc` function to access fields. Also note the use of Augmented assignment statements (`*=`) to modify data in-place.:

```
def scenario_co2_limit(data):
    # change global CO2 limit
    hacks = data['hacks']
    hacks.loc['Global CO2 limit', 'Value'] *= 0.05
    return data
```

Scenario `scenario_co2_limit()` shows the simple case of changing a single input data value. In this case, a 95% CO2 reduction compared to the base scenario must be accomplished. This drastically limits the amount of coal and gas that may be used by all three sites.:

```
def scenario_north_process_caps(data):
    # change maximum installable capacity
    pro = data['process']
    pro.loc[('North', 'Hydro plant'), 'cap-up'] *= 0.5
    pro.loc[('North', 'Biomass plant'), 'cap-up'] *= 0.25
    return data
```

Scenario `scenario_north_process_caps()` demonstrates accessing single values in the `process` `DataFrame`. By reducing the amount of renewable energy conversion processes (hydropower and biomass), this scenario explores the "second best" option for this region to supply its demand.:

```
def scenario_all_together(data):
    # combine all other scenarios
    data = scenario_stock_prices(data)
    data = scenario_co2_limit(data)
    data = scenario_north_process_caps(data)
    return data
```

Scenario `scenario_all_together()` finally shows that scenarios can also be combined by chaining other scenario functions, making them dependent. This way, complex scenario trees can written with any single input change coded at a single place and then building complex composite scenarios from those.

Run scenarios

This now finally is the function that gets everything going. It is invoked in the very end of the runscript.

```
for scenario in scenarios:
prob = urbs.run_scenario(input_files, solver, timesteps, scenario,
                     result_dir, dt, objective,
                     plot_tuples=plot_tuples,
                     plot_sites_name=plot_sites_name,
                     plot_periods=plot_periods,
                     report_tuples=report_tuples,
                     report_sites_name=report_sites_name)
```

Having prepared settings, input data and scenarios, the actual computations happen in the function `run_scenario()` of the script `runfunctions.py` in subfolder `urbs`. It is executed for each of the scenarios included in the scenario list. The following sections describe the content of function `run_scenario()`. In a nutshell, it reads the input data from its argument `input_file`, modifies it with the supplied `scenario`, runs the optimisation for the given `timesteps` and writes report and plots to `result_dir`.

Reading input

```
# scenario name, read and modify data for scenario
sce = scenario.__name__
data = read_input(input_files,year)
data = scenario(data)
validate_input(data)
```

Function `read_input()` returns a dict `data` of up to 12 pandas DataFrames with hard-coded column names that correspond to the parameters of the optimization problem (like `eff` for efficiency or `inv-cost-c` for capacity investment costs). The row labels on the other hand may be freely chosen (like site names, process identifiers or commodity names). By convention, it must contain the six keys `commodity`, `process`, `storage`, `transmission`, `demand`, and `supim`. Each value must be a `pandas.DataFrame`, whose index (row labels) and columns (column labels) conforms to the specification given by the example dataset in the spreadsheet `mimo-example.xlsx`.

`data` is then modified by applying the `scenario()` function to it. To then rule out a list of known errors, that accumulate through growing user experience, a variety of validation functions specified in script `validate.py` in subfolder `urbs` is run on the dict `data`.

Solving

```
# create model
prob = urbs.create_model(data, dt, timesteps)

# refresh time stamp string and create filename for logfile
now = prob.created
log_filename = os.path.join(result_dir, '{}.log').format(sce)

# solve model and read results
optim = SolverFactory('glpk')  # cplex, glpk, gurobi, ...
```

(continues on next page)

(continued from previous page)

```
optim = setup_solver(optim, logfile=log_filename)
result = optim.solve(prob, tee=True)
```

This section is the "work horse", where most computation and time is spent. The optimization problem is first defined (`create_model()`) and populated with parameter values with values. The `SolverFactory` object is an abstract representation of the solver used. The returned object `optim` has a method `set_options()` to set solver options (not used in this tutorial).

The remaining line calls the solver and reads the `result` object back into the `prob` object, which is queried to for variable values in the remaining script file. Argument `tee=True` enables the realtime console output for the solver. If you want less verbose output, simply set it to `False` or remove it.

Business park example explained

In this part the input files of the standard example **Business park** will be explained in detail.

Task

The task we set ourselves here is to build our own intertemporal model. The task is the following:

The technical staff of a business park management company wants you to find the cost optimal energy system for their business park. You are to provide this with increasingly stricter CO2 emission limits over time. As the company expects to operate this business park for a long time still, they want you to help developing a long term strategy how to transform the energy supply infrastructure of the business park in cost optimal way over the time frame of 3 decades. The company also expects that the business park will be increasingly closely interacting with the neighboring small city and its energy system. All current and expected demand curves are given to you. You also have full access to regional climate models and all relevant parameters for the energy conversion units relevant for your problem.

Input files

The task set is intertemporal. That is we need to provide several .xlsx input files, one for each modeled year. Here we chose to use 3 files representing modeled years 10 years apart. For the given task this seems to be a good compromise between accuracy and computational effort. The files are named `2020.xlsx`, `2030.xlsx` and `2040.xlsx` and sit in the folder `Input (Business park)`. We will now proceed with a detailed walkthrough of the individual files.

Sheet Global

Here you can now specify the global properties needed for the modeling of the energy system. Note that this sheet has different entries for the different input files:

- **Support timeframe** (All files): Give the value for the modeled year here.
- **Discount rate** (Only first file): This value gives the discount rate that is used for intertemporal planning. It stands for the annual devaluation of money across the modeling horizon. In the example a discount rate of *3 %* is used.

- **CO2 limit** (All files): This parameter limits the CO2 emissions across all sites within one modeled year, the *CO2 budget* sets a cap on the total emissions across all sites in the entire modeling horizon. If no restriction is desired enter 'inf' here. In the example increasingly strict values for the CO2 limit are used for the different modeled years, from *60 kt/a* in 2020 over *45 kt/a* in 2030 to *30 kt/a* in 2040. This represents the will of the company to achieve milestones in the emission reductions while gradually changing their energy infrastructure.
- **CO2 budget** (Only first file): While the *CO2 limit* specified for each year limits the CO2 emissions across all sites within one modeled year, the *CO2 budget* sets a cap on the total emissions across all sites in the entire modeling horizon. If no restriction is desired enter 'inf' here. The *CO2 budget* is only active when the *Objective* is set to its default value 'cost'. In the example a CO2 budget of *1.2 Mt* is used. This budget imposes a stricter limit on the emissions than the combined targets for the individual modeled year. In terms of climate impact his limit is the more important one. For all CO2 limitations the business park and the city are considered together since in the assumed case the company running the business park wants to act as an electricity provider for the city as well.
- **Cost budget** (Only first file): With this parameter a limit on the total system cost over the entire modeling horizon can be set. If no restriction is desired enter 'inf' here. The *Cost budget* is only active when the *Objective* is set to the value 'CO2'. In the example no CO2 optimization is considered this parameter is thus set to infinity.
- **Last year weight** (Only last file): In intertemporal modeling each modeled year is repeated until the next modeled year is reached. This is done ba assigning a weight to the costs accrued in each of the modeled years. For the last modeled year the number of repetitions has to be set by the user here, where a high number leads to a stronger weighting of the last modeled year, i.e. of the final energy system configuration. In the example the last year has a weight of *10 years*. This means that it will be equally weighted identically to the others which always represent all years until the ext modeled year.

Sheet Site

In this sheet you can specify the site names and also the area of each site. The line index represents all the sites. The only site specific property to be set is then:

- **Area**: Specifies the usable area for processes in the given site. The area does not need to be the total floor area. It is used to limit the use of area consuming processes and can be seen as, e.g., the roof area for solar technologies.

  In the example two sites 'Business park' and 'City' are given. These and their respective areas do not change. The areas here represent roof areas for PV and the city has more of this.

Sheet Commodity

In this sheet all the commodities, i.e. energy or material carriers, are specified. The line index completes a commodity tuple, i.e. a connection `(year, site, commodity, type)`. There are three properties to be specified for all commodities of types **Stock**, **Buy**, **Sell** and **Environmental**.

- **Price** denotes the cost of taking one unit of energy from the stock for **Stock** commodities or emitting one unit of **Environmental**. For **Buy** and **Sell** commodities this is not directly a price but a multiplier for the time series given in the sheet 'Buy-Sell-Price'. It is thus typically set to 1 for these commodity types.

- **max** limits the total amount of the commodity that may be bought, sold or emitted per year.
- **maxperhaour** limits the total amount of the commodity that may be bought, sold or emitted per hour (not timestep but really hour).

In the site 'Business park' there are 9 commodities defined:

- *Solar (West/East)* is of type **SupIm** and represents the capacity factor timeseries of solar panels mounted with a given inclination (10° both West and East).
- *Grid electricity* is of type **Buy** and represents the electricity price as bought from the regional grid operator. The business park pays constant price over the year. In the site 'City' this price is different and hence a multiplier is used to increase the wholesale price for households.
- *Gas* is of type **Stock** and represents the price for the purchase of natural gas from the local provider.
- *Electricity*, *Heat* and *Cooling* are of type **Demand** and represent the hourly demand for these three energy carriers.
- *Intermediate* is of type **Stock**. However, it is not possible to buy this commodity from the stock. It is introduced to allow for a flexible operation of a combined heat and power (CHP) plant according to section *Modeling nuggets*.
- *Intermediate low temperature* is of type **Stock**. It is also not buyable from an external source. Its purpose is to make the operation of the cooling processes more realistic by preventing the storage of high temperature cooling from ambient air cooling in cold storages.

In site 'City' one additional commodity, *Operation decentral units* is introduced. It is of type **SupIm** and makes sure that the different heating technologies usable in the site all operate at a fixed share of the total heat demand. This is necessary, since these technologies are build up in a decentral way in the individual houses. The idea behind this is laid out in section *Modeling nuggets*.

Sheet Process

In this sheet the energy conversion technologies are described. Here only the economical and some general technical parameters are set. The interactions with the commodities are introduced in the next sheet. The following parameters are set here for the processes:

- **Installed capacity (MW) (Only first file)** gives the capacity of the process that is already istalled at the start of the modeling horizon.
- **Lifetime of installed capacity (years) (Only first file)** gives the rest lifetime of the installed processes in years. A process can be used in a modeled year *y* still if the lifetime plus the first modeled year exceeds the next year *y+1*.
- **Minimum capacity (MW)** denotes a capacity target that has to be met by the process in a given modeled year. This means that the system will build at least this capacity.
- **Maximum capacity (MW)** restricts the capacity that can be built to the specified value.
- **Maximum power gradient (1/h)** restricts the ramping of process operational states, i.e. the change in the throughput variable. The value gives the fraction of the total capacity that can be changed in one hour. A value of *1* thus restricts the change from idle to full operational state (or vice versa) to at least a duration of one hour.

- **Minimum load fraction** gives a lower limit for the operational state of a process as a fraction of the total capacity. It is only relevant for processes where part-load behavior is modeled. A value here is only active when 'Ratio-Min' is numerical for at least one input commodity.
- **Investment cost (€/MW)** denotes the capacity specific investment costs for the process. You should give the book value here. The program will then translate this into the correct total, discounted cost within the model horizon.
- **Annual fix costs (€/MW)** represent the amount of money that has to be spent annually for the operation of a process capacity. They can represent, e.g., labour costs or calendaric ageing costs.
- **Variable costs (€/MWh)** are linked to the operation of a process and are to be paid for each unit of throughput through the process. They can represent anything from usage ageing to taxes.
- **Weighted average cost of capital** denotes the interest rate or expected return on investment with which the investor responsible for the energy system calculates.
- **Depreciation period** denotes both the economical and technical lifetime of all units in the system. It thus determines two things: the total costs of a given investment and the end of operational time for all units in the energy system modeled.
- **Area use per capcacity (m^2/MW)** specifies the physical area a given process takes up at the site it is built. This can be used, e.g. to restrict the capacity of solar technologies by a total maximal roof area. The restricting area is defined in sheet 'Site'.

While the details of the processes will be discussed in more detail in the next section one mention on the processes 'Load dump' and 'Slack' is made here. These processes are not introduced to represent real units but help making operation more realistic and error fixing more easy. A load dump process just destroys energy which is sometimes necessary in order to prevent the system from doing unrealistic gymnastics to keep the vertex rule. A 'Slack' process can create a demand commodity out of thin air for an extremely high price. It thus indicates when the problem is not feasible, making error fixing much easier. Both should typically be included in models.

Sheet Process-Commodity

In this sheet the technical properties of processes are set. These properties are given for each process independent of the site where the process is located. You need to make an imput for all the processes defined in the 'Process' sheet. The line index is a tuple `(process, commodity, direction)`, where 'Direction' has to be set as either 'In' or 'Out' and specifies wether a commodity is an in- or an output of a given process. Under the column 'ratio' you then have to specify the commodity in- or outflows per installed capacity and time step at the point of full operation. The efficiency of the process for the conversion of one input into one output commodity is then given by the ratio of the chosen values. For example, in the modeled year 2020 the process 'Gas engine power plant' converts 2.2 MWh of 'Gas' into one MWh each of 'Electricity' and 'Heat' while emitting 0.2 t of 'CO2'. This corresponds to an efficiency of 0.45 for 'Heat' and 'Electricity' conversion.

If a process has a more complex part load behavior, where, e.g., the efficiency changes this can be partly captured by setting values in the 'ratio-min' column. These specify the commodity flows at the minimum operation point specified in the 'Process' sheet under 'min-fract'. The process will then no longer be allowed to turn off so use this carefully. In the present case this behavior is set for the combined heat and power plant 'CHP (Operational state)' only.

There are a few special settings made in the example. First, the CHP plant is divided into three parts. The idea behind this is described in *Modeling nuggets*. The two processes 'CHP (Electricity driven)' and 'CHP (Heat driven)' specify the commodity flows in the two extreme operational states. The system can

then chose all linear interpolations between both states by guiding the commodity flow of 'Intermediate' through the two processes in the desired ratio. Second, the cooling technologies are implemented in a two stage way. The reason for this is that the process 'Ambient air cooling' is extremely efficient and extremely cheap. While it can only be used in certain time intervals (see explanation of 'TimeVarEff' further below), its output could nevertheless be stored otherwise which is not realistic. The introduction of commodity 'Intermediate low temperature' prevents this. It is the output of all the cooling technologies except for 'Ambient air cooling' and is also the one that can be stored (see below).

Sheet Transmission

In this sheet the parameters for transmission lines between sites are specified. The line index is part of a transmission tuple `(Site In, Site Out, Transmission, Commodity)`. Note that for each transmission the inverse one with the same properties should also be given. The parameters are the following:

- **Efficiency (1)** specifies the transport efficiency of the transmission line.
- **Lifetime of installed capacity (years) (Only first file)** gives the rest lifetime of the installed transmission lines in years. A transmission line can be used in a modeled year *y* still if the lifetime plus the first modeled year exceeds the next year *y+1*.
- **Investment cost (€/MW)** denotes the capacity specific investment costs for the transmission line. You should give the book value here. The program will then translate this into the correct total, discounted cost within the model horizon.
- **Annual fix costs (€/MW)** represent the amount of money that has to be spent annually for the operation of a transmission capacity. They can represent, e.g., labour costs or calendaric ageing costs.
- **Variable costs (€/MWh)** are linked to the operation of a given transmission line.
- **Installed capacity (MW) (Only first file)** gives the transmission capacity of transmission lines already installed at the start of the modeling horizon.
- **Minimum capacity (MW)** denotes a transmission capacity target that has to be met by the transmission lines in a given modeled year. This means that the system will build at least this transmission capacity.
- **Maximum capacity (MW)** restricts the transmission capacity that can be built to the specified value.
- **Weighted average cost of capital** denotes the interest rate or expected return on investment with which the investor responsible for the energy system calculates.
- **Depreciation period** denotes both the economical and technical lifetime of all units in the system. It thus determines two things: the total costs of a given investment and the end of operational time for all units in the energy system modeled.

In the example the only commodity that can be transported from one site to the other is electricity.

Sheet Storage

In this sheet the parameters for storage units are specified. Each storage unit is indexed with parts of a storage tuple `(storage, commodity)`. In storages charging/discharging power and the capacity are sized independently. The parameters specifying the storage properties are the following:

- **Installed capacity (MWh) (Only first file)** gives the storage capacity of storages already installed at the start of the modeling horizon.
- **Installed storage power (MW) (Only first file)** gives the charging/discharging power of storages already installed at the start of the modeling horizon.
- **Lifetime of installed capacity (years) (Only first file)** gives the rest lifetime of the installed storages in years. A storage can be used in a modeled year *y* still if the lifetime plus the first modeled year exceeds the next year *y+1*.
- **Minimum storage capacity (MWh)** denotes a storage capacity target that has to be met by the storage in a given modeled year. This means that the system will build at least this capacity.
- **Maximum storage capacity (MWh)** restricts the storage capacity that can be built to the specified value.
- **Minimum storage power (MW)** denotes a storage charging/discharging power target that has to be met by the storage in a given modeled year. This means that the system will build at least this power.
- **Maximum storage power (MW)** restricts the storage charging/discharging that can be built to the specified value.
- **Efficiency input (1)** specifies the charging efficiency of the storage.
- **Efficiency output (1)** specifies the discharging efficiency of the storage.
- **Investment cost capacity (€/MWh)** denotes the storage capacity specific investment costs for the storage. You should give the book value here. The program will then translate this into the correct total, discounted cost within the model horizon.
- **Investment cost power (€/MW)** denotes the storage charging/discharging power specific investment costs for the storage. You should give the book value here. The program will then translate this into the correct total, discounted cost within the model horizon.
- **Annual fix costs capacity (€/MWh)** represent the amount of money that has to be spent annually for the operation of a storage capacity. They can represent, e.g., labour costs or calendaric ageing costs.
- **Annual fix costs power (€/MW)** represent the amount of money that has to be spent annually for the operation of a storage power. They can represent, e.g., labour costs or calendaric ageing costs.
- **Variable costs capacity (€/MWh)** are linked to the operation of a given storage state, i.e. they lead to costs whenever a storage has a non-zero state of charge. These costs should typically set to zero but can be used to manipulate the storage duration or model state-of-charge dependent ageing.
- **Variable costs power (€/MWh)** are linked to the charging and discharging of a storage. Each unit of commodity leaving the storage requires the payment of these costs.
- **Weighted average cost of capital** denotes the interest rate or expected return on investment with which the investor responsible for the energy system calculates.
- **Depreciation period** denotes both the economical and technical lifetime of all units in the system. It thus determines two things: the total costs of a given investment and the end of operational time for all units in the energy system modeled.
- **Initial storage state** can be used to set the state of charge of a storages in the beginning of the modeling time steps. If *nan* is given this value is an optimization variable. In any case the storage

content in the end is the same as in the beginning to avoid windfall profits from simply discharging a storage.

- **Discharge** gives the hourly discharge of a storage. Over time, when no charging or discharging occurs, the storage content will decrease exponentially.

In the example there are no storages in site 'City' and there is a storage for each demand in site 'Business park'. The commodity 'Cooling' is not directly storable to avoid an unrealistic behavior for the process 'Ambient air cooling' as was discussed above in the 'Process-Commodity' section.

Sheets Demand, SupIm, Buy/Sell price

In these sheets the time series for all the demands, capacity factors of processes using commodities of type 'SupIm' and market prices for 'Buy' and 'Sell' commodities are to be specified. For the former two the syntax 'site.commodity' has to be used as a column index to specify the corresponding tuple.

Sheet TimeVarEff

In this sheet a time series for the output of processes can be given. This is always useful, when processes are somehow dependent on external parameters. The syntax to be used to specify which process is to be addressed by this is 'site.process'. In the present example, this is used for the process 'Ambient air cooling' which has a boolean 'TimeVarEff' curve giving the value '1' for temperatures below a threshold and '0' else.

This concludes the input generation. Of course all parameters have to be set in all the input sheets.

Run script

To run the example you can make a copy of the file `runme.py` calling it, e.g., `run_BP.py` in the same folder. You now just have to make 3 modifications. First, replace the report tuples by:

```
report_tuples = [
    (2020, 'Business park', 'Electricity'),
    (2020, 'Business park', 'Heat'),
    (2020, 'Business park', 'Cooling'),
    (2020, 'Business park', 'Intermediate low temperature'),
    (2020, 'Business park', 'CO2'),
    (2030, 'Business park', 'Electricity'),
    (2030, 'Business park', 'Heat'),
    (2030, 'Business park', 'Cooling'),
    (2030, 'Business park', 'Intermediate low temperature'),
    (2030, 'Business park', 'CO2'),
    (2040, 'Business park', 'Electricity'),
    (2040, 'Business park', 'Heat'),
    (2040, 'Business park', 'Cooling'),
    (2040, 'Business park', 'Intermediate low temperature'),
    (2040, 'Business park', 'CO2'),
    (2020, 'City', 'Electricity'),
    (2020, 'City', 'Heat'),
    (2020, 'City', 'CO2'),
    (2030, 'City', 'Electricity'),
    (2030, 'City', 'Heat'),
```

(continues on next page)

(continued from previous page)

```
    (2030, 'City', 'CO2'),
    (2040, 'City', 'Electricity'),
    (2040, 'City', 'Heat'),
    (2040, 'City', 'CO2'),
    (2020, ['Business park', 'City'], 'Electricity'),
    (2020, ['Business park', 'City'], 'Heat'),
    (2020, ['Business park', 'City'], 'CO2'),
    (2030, ['Business park', 'City'], 'Electricity'),
    (2030, ['Business park', 'City'], 'Heat'),
    (2030, ['Business park', 'City'], 'CO2'),
    (2040, ['Business park', 'City'], 'Electricity'),
    (2040, ['Business park', 'City'], 'Heat')
    (2040, ['Business park', 'City'], 'CO2'),
    ]

# optional: define names for sites in report_tuples
report_sites_name = {('Business park', 'City'): 'Together'}
```

and the plot tuples by:

```
plot_tuples = [
    (2020, 'Business park', 'Electricity'),
    (2020, 'Business park', 'Heat'),
    (2020, 'Business park', 'Cooling'),
    (2020, 'Business park', 'Intermediate low temperature'),
    (2020, 'Business park', 'CO2'),
    (2030, 'Business park', 'Electricity'),
    (2030, 'Business park', 'Heat'),
    (2030, 'Business park', 'Cooling'),
    (2030, 'Business park', 'Intermediate low temperature'),
    (2030, 'Business park', 'CO2'),
    (2040, 'Business park', 'Electricity'),
    (2040, 'Business park', 'Heat'),
    (2040, 'Business park', 'Cooling'),
    (2040, 'Business park', 'Intermediate low temperature'),
    (2040, 'Business park', 'CO2'),
    (2020, 'City', 'Electricity'),
    (2020, 'City', 'Heat'),
    (2020, 'City', 'CO2'),
    (2030, 'City', 'Electricity'),
    (2030, 'City', 'Heat'),
    (2030, 'City', 'CO2'),
    (2040, 'City', 'Electricity'),
    (2040, 'City', 'Heat'),
    (2040, 'City', 'CO2'),
    (2020, ['Business park', 'City'], 'Electricity'),
    (2020, ['Business park', 'City'], 'Heat'),
    (2020, ['Business park', 'City'], 'CO2'),
    (2030, ['Business park', 'City'], 'Electricity'),
    (2030, ['Business park', 'City'], 'Heat'),
    (2030, ['Business park', 'City'], 'CO2'),
    (2040, ['Business park', 'City'], 'Electricity'),
    (2040, ['Business park', 'City'], 'Heat')
    (2040, ['Business park', 'City'], 'CO2'),
    ]
```

(continues on next page)

(continued from previous page)

```
# optional: define names for sites in plot_tuples
plot_sites_name = {('Business park', 'City'): 'Together'}
```

In this way you get a meaningful output for the optimization runs. Second, the scenarios are made for the other example and are as such no longer usable here. Thus only the base scenario is to be run. Change the list scenario to the following:

```
scenarios = [
             urbs.scenario_base
            ]
```

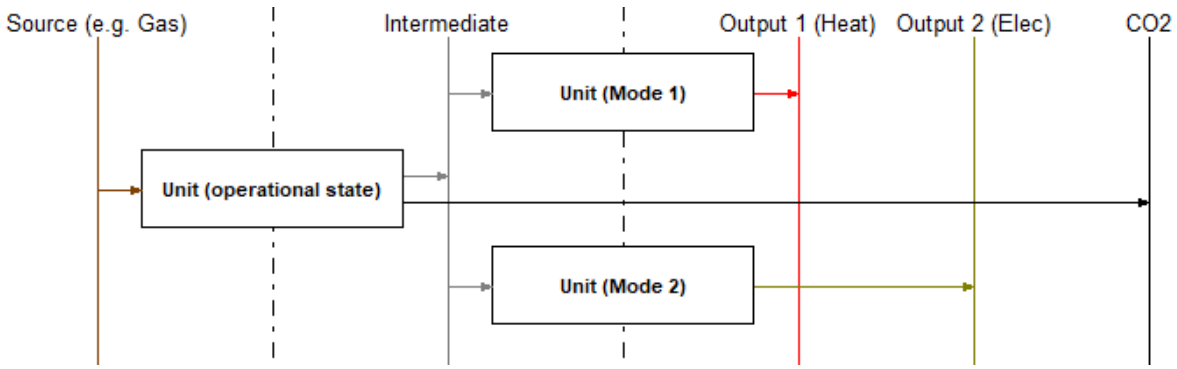
Having completed all these steps you can execute the code.

Modeling nuggets

Here you can find a collection of non-trivial modeling ideas that can be used in linear energy system modeling with urbs. It is meant for more advanced users and you should fully understand the two standard examples **mimo-example** and **Business park** before proceeding. What follows is a loose collection of modeling approaches and does not follow any internal logic.

Different operational modes

For many power plants as, e.g., combined heat and power plants (CHP) there are different modes of operation. These and intermediate states between the extremes can be well captured in urbs models using the approach sketched in the following picture:



Here the vertical lines represent the commodities and the rectangle are processes. The arrows indicate in- and output commodities of the processes. In the case shown the power plant 'Unit' would be able to operate between a state where only 'Output 1' comes out and a state where only 'Output 2' comes out. The two extreme cases can, however, also be chosen as combinations of both outputs already.

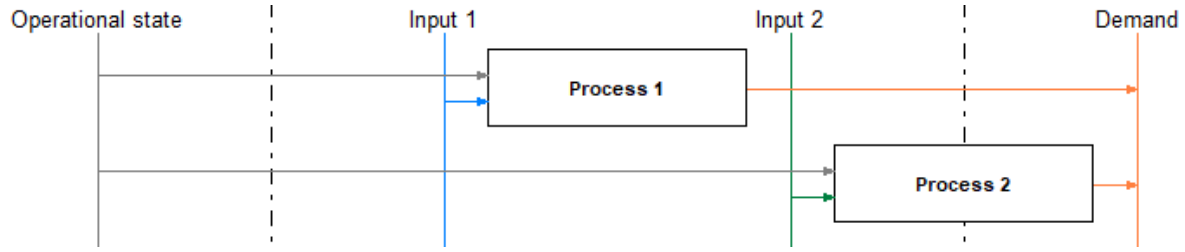
The idea behind the figure is the following: The commodity 'Intermediate' is to be produced exclusively by the process 'Unit (operational state)'. It thus simply tracks the throughput of this process. Due to the vertex rule (Kirchhoff´s current law) the commodity 'Intermediate' once produced needs to be consumed immediately. This can happen either via 'Unit (Mode 1)', 'Unit (Mode 2)' or a linear combination of both. The result is then the desired choice for the optimizer between states formed by linear combinations of the two modes. The commodity 'Intermediate' is best chosen as a **Stock** commodity where either the price is set to infinity or the maximum allowed usage per hour, or year (or both) is set to zero. This ensures that the commodity has to be produced by the process and cannot be bought from an external source, which for the present case would of course be absurd.

All process parameters and the setting of part load, time variable efficiency etc. is best done for the 'Unit (operational state)' process. The two other processes should in turn be used as mathematical entities that are defined by their 'process commodity' input only.

Proportional operation

Often many individual consumers are lumped together in one site. If a demand of these consumers is then met by a collection of decentral units it is important that the different technology options for these decentral units each fulfill a fixed fraction of the demand in each time step. This means that the different technology options are proportional to each other and the demand.

This behavior can be enforced by the following design:



Here the vertical lines represent the commodities and the rectangle are processes. The arrows indicate in- and output commodities of the processes.

For the desired result the commodity 'Operational state' has to be of type **SupIm** and the corresponding time series has to be set as the normalized demand. in this way the optimizer can still size the two technology options 'Process 1' and 'Process 2' optimally while being forced to operate them proportionally to each other and to the demand. Other input or output (not shown) commodities can then be associated with the process operation as usual and will be dragged along by the forced operation.

Scenario generation

For a sensitivity analysis, it might be helpful to not manually create all scenario definitions automatically. For example, if one is interested in how installed capacities of PV and storage change the output, one might define ranges for each capacity. If there are four thresholds for the PV capacity and five for storage capacity, creating all 20 scenarios by hand is quite tiresome.

In this example, one wants to run an optimization with capacities 20 GW, 30 GW, 40 GW and 50 GW for PV and 50 GW, 60 GW, 70 GW, 80 GW and 90 GW for storage capacities.

Therefore, a function factory is created, which takes the values for PV and storage capacity and creates a scenario function out of it. This is done in the file *scenarios.py*:

```
def create_scenario_pv_sto(pv_val, sto_val):
    def scenario_pv_sto(data):
        # set PV capacity for all sites
        pro = data['process']
        solar = pro.index.get_level_values('Process') == 'Photovoltaics'
        pro.loc[solar, 'inst-cap'] = pv_val
        pro.loc[solar, 'cap-up'] = sto_val

        # set storage content capacity
        sto = data['storage']
```

(continues on next page)

(continued from previous page)

```
        for site_sto_tuple in sto.index:
            sto.loc[site_sto_tuple, 'inst-cap-c'] = sto_val
            sto.loc[site_sto_tuple, 'cap-up-c'] = sto_val

        return data
    # define name for scenario dependent on pv and storage values
    scenario_pv_sto.__name__ = f"scenario_pv{int(pv_val/1000)}_sto{int(sto_
↪val/1000)}"
    return scenario_pv_sto
```

In *runme.py* the following has to be added:

```
# define range for sensitvity
pv_vals = range(20000, 50001, 10000)
sto_vals = range(50000, 90001, 10000)

# create scenario functions
scenarios = []
for pv_val in pv_vals:
    for sto_val in sto_vals:
        scenarios.append(urbs.create_scenario_pv_sto(pv_val, sto_val))
```

1.2 Mathematical documentation

Continue here if you want to understand the theoretical conception of the model generator, the logic behind the equations and the structure of the features.

1.2.1 Mathematical description

In this Section the **mathematical description** of a model generated by urbs will be explained. The structure here follows the basic code structure and proceeds as follows:

First, a short introduction into the type of optimization problem solvable with urbs is given. This is followed by the description of the minimal possible model in urbs. As a next step the two main expansions of models, which also increase the index depth of all variables and parameters are discussed in the parts 'Intertemporal modeling' and 'Multinode modeling'. The description is then concluded by the additional description of various feature modules. The latter are discussed in full index depth, i.e., with all features introduced in minimal, intertemporal and multinode modeling.

Structure of an urbs model

urbs is an abstract generator for linear optimization problems. Such problems can in general be written in the following standard form:

$$\begin{aligned} \min\ & c^{\mathrm{T}} x \\ \text{s.t. } & Ax = b \\ & Bx \leq d. \end{aligned}$$

where x is the variable vector, c the coefficient vector for the objective function and A and B the matrices for the equality and inequality constraints, respectively. The equality constraints could also

be represented by inequality constraints, which is not done here for simplicity reasons. There are two options for the objective function: either the total system costs or environmental emissions can be used. The structure of the following parts will be first a description of x and c and subsequently a general formulation of the constraint functions that make up the matrices A and B as well as the vectors b and d. All variables and equations will be first presented for a minimally complex problem and the optional additional variables and equations are presented in extra parts.

Energy system entities

For all models that can be generated with urbs, the energy system is built up out of the following entities:

- Commodities, which represent the various forms of material and energy flows in the system.
- Processes, which convert commodities from one type to another. These entities are always multiple-input/multiple-output (mimo) that is, a certain fixed set of input commodities is converted into another fixed set of output commodities.
- Transmission lines, that allow for the transport of commodities between the modeled spatial vertices.
- Storages, which allow the storage of a single type of commodity.
- DSM potentials, which make the time shifting of demands possible.

Minimal optimization model

The minimal model in urbs is a simple expansion and dispatch model with only processes being able to fulfill the given demands. All spatial information is neglected in this case. The minimal model is already multiple-input/multiple output (mimo) and the variable vector takes the following form:

$$x^{\mathrm{T}} = (\zeta, \underbrace{\rho_{ct}}_{\text{commodity variables}}, \underbrace{\kappa_p, \widehat{\kappa}_p, \tau_{pt}, \epsilon^{\text{in}}_{cpt}, \epsilon^{\text{out}}_{cpt}}_{\text{process variables}}).$$

Here, ζ represents the total annualized system cost, $\rho_c t$ the amount of commodities c taken from a virtual, infinite stock at time t, κ_p and $\widehat{\kappa}_p$ the total and the newly installed process capacities of processes p, τ_{pt} the operational state of processes p at time t and $\epsilon^{\text{in}}_{cpt}$ and $\epsilon^{\text{out}}_{cpt}$ the total inputs and outputs of commodities c to and from process p at time t, respectively.

Objective

For any urbs problem as the objective function either the total system costs or the total emissions of CO2 can be chosen. In the former (standard) case this leads to an objective vector of:

$$c = (1, 0, 0, 0, 0, 0, 0),$$

where only the costs are part of the objective function. For the latter choice of objective no such simple structure can be written.

Costs

In the minimal model the total cost variable can be split into the following sum:

$$\zeta = \zeta_{\text{inv}} + \zeta_{\text{fix}} + \zeta_{\text{var}} + \zeta_{\text{fuel}} + \zeta_{\text{env}},$$

where ζ_{inv} are the annualized invest costs, ζ_{fix} the annual fixed costs, ζ_{var} the total variable costs accumulating over one year, ζ_{fuel} the accumulated fuel costs over one year and ζ_{env} the annual penalties for environmental pollution. These costs are then calculated in the following way:

Annualized invest costs

Investments are typically depreciated over a longer period of time than the standard modeling horizon of one year. To overcome distortions in the overall cost function urbs uses the annual cash flow (CAPEX) for the calculation of the investment costs in the cost function. This is captured by multiplying the total invest costs for a given process C_p with the so-called annuity factor f_p, i.e.:

$$\zeta_{\text{inv},p} = f_p \cdot C_p$$

For an interest rate of i and a depreciation period of n years the annuity factor can be derived using the remaining debt after k payments C_k:

After 0 Payments: $C_0 = C(1+i)$

After 1 Payment: $C_1 = (C_0 - fC)(1+i) = C(1+i)^2 - fC(1+i)$

After 2 Payments: $C_2 = (C_1 - fC)(1+i) = C(1+i)^3 - fC(1+i)^2 - fC(1+i)$

...

After n Payments: $C_n = C(1+i)^n + C\sum_{k=0}^{n-1}(1+i)^k = (1+i)^n + f\left(\frac{1-(1+i)^n}{i}\right).$

Since the outstanding debt becomes 0 at the end of the depreciation period this leads to:

$$f = \frac{(1+i)^n \cdot i}{(1+i)^n - 1}$$

The annualized invest costs for all investments made by the optimizer are then given by:

$$\zeta_{\text{inv}} = \sum_{p \in P_{\text{exp}}} f_p k_p^{\text{inv}} \widehat{\kappa}_p,$$

where k_p^{inv} signifies the specific invest costs of process p per unit capacity and P_{exp} is the subset of all processes that are actually expanded.

Annual fixed costs

The annual fixed costs represent maintenance and staff payments the processes used. They are playing a role for unit expansion only and are given as parameters for all allowed processes. Fixed costs scale with the capacity (in W) of the processes, and can be calculated using:

$$\zeta_{\text{fix}} = \sum_{p \in P} k_p^{\text{fix}} \kappa_p,$$

where k_p^{fix} represents the specific annual fix costs for process p.

Annual variable costs

Variable costs represent both, additional maintenance requirements due to usage of processes and taxes or tariffs. They scale with the utilization of processes (in Wh) and can be calculated in the following way:

$$\zeta_{\text{var}} = w\Delta t \sum_{t \in T_m} p \in P k_{pt}^{\text{var}} \tau_{pt},$$

where k_{pt}^{var} are the specific variable costs per time integrated process usage, and w and Δt are a weight factor that extrapolates the actual modeled time horizon to one year and the timestep length in hours, respectively.

Annual fuel costs

The usage of fuel adds an additional cost factor to the total costs. As with variable costs these costs occur when processes are used and are dependent on the total usage of the fuel (stock) commodities:

$$\zeta_{\text{fuel}} = w\Delta t \sum_{t \in T_m} c \in C_{\text{stock}} k_c^{\text{fuel}} \rho_c,$$

where k_c^{fuel} are the specific fuel costs. The distinction between variable and fuel costs is introduced for clarity of the results, both could in principle be merged into one class of costs.

Annual environmental costs

Environmental costs occur when the emission of an environmental commodity is penalized by a fine. Environmental commodities do not have to be balanced but can be emitted to the surrounding. The total production of the polluting environmental commodity is then given by:

$$\zeta_{\text{env}} = -w\Delta t \sum_{t \in T_m} c \in C_{\text{env}} k_c^{\text{env}} \text{CB}(c,t),$$

where k_c^{env} are the specific costs per unit of environmental commodity and CB is the momentary commodity balance of commodity c at time t. The minus sign is due to the sign convention used for the commodity balance which is positive when the system takes in a unit of a commodity.

After this discussion of the individual cost terms the constraints making up the matrices A and B are discussed now.

Process expansion constraints

The unit expansion constraints are independent of the modeled time. In case of the minimal model the are restricted to two constraints only limiting the allowed capacity expansion for each process. The total capacity of a given process is simply given by:

$$\forall p \in P :$$
$$\kappa_p = K_p + \widehat{\kappa}_p,$$

where K_p is the already installed capacity of process p. The newly installed capacity can also be an integer, expressed as the product between the parameter process new capacity block K_p^{block} and the variable new process capacity units β_p:

$$\widehat{\kappa}_p = K_p^{\text{block}} \cdot \beta_p$$

Process capacity limit rule

The capacity of each process p is limited by a maximal and minimal capacity, $\overline{K}_p$ and $\underline{K}_p$, respectively, which are both given to the model as parameters:

$$\forall p \in P :$$
$$\underline{K}_p \leq \kappa_p \leq \overline{K}_p.$$

All further constraints are time dependent and are determinants of the unit commitment, i.e. the time series of operation of all processes and commodity flows.

Commodity dispatch constraints

In this part the rules governing the commodity flow timeseries are shown.

Vertex rule ("Kirchhoffs current law")

This rule is the central rule for the commodity flows and states that all commodity flows, (except for those of environmental commodities) have to be balanced in each time step. As a helper function the already mentioned commodity balance is calculated in the following way:

$$\forall c \in C,\ t \in T_m :$$

$$\text{CB}(c,t) = \sum_{(c,p)\in C_p^{\text{out}}} \epsilon_{cpt}^{\text{in}} - \sum_{(c,p)\in C_p^{\text{in}}} \epsilon_{cpt}^{\text{out}}.$$

Here, the tuple sets $C_p^{\text{in,out}}$ represent all input and output commodities of process p, respectively. The commodity balance thus simply calculates how much more of commodity c is emitted by than added to the system via process p in timestep t. Using this term the vertex rule for the various commodity types can now be written in the following way:

$$\forall c \in C_{\text{st}},\ t \in T_m : \rho_{ct} \geq \text{CB}(c,t),$$

where C_{st} is the set of stock commodities and:

$$\forall c \in C_{\text{dem}},\ t \in T_m : -d_{ct} \geq \text{CB}(c,t),$$

where C_{dem} is the set of demand commodities and d_{ct} the corresponding demand for commodity c at time t. These two rules thus state that all stock commodities that are consumed at any time in any process must be taken from the stock and that all demands have to be fulfilled at each time step.

Stock commodity limitations

There are two rule that govern the retrieval of stock commodities from stock: The total stock and the stock per hour rule. The former limits the total amount of stock commodity that can be retrieved annually and the latter limits the same quantity per timestep. the two rules take the following form:

$$\forall c \in C_{\text{st}}:$$
$$w \sum_{t \in T_m} \rho_{ct} \leq \overline{L}_c$$

$$\forall c \in C_{\text{st}},\ t \in T_m:$$
$$\rho_c t \leq \bar{l}_c,$$

where $\overline{L}_c$ and $\bar{l}_c$ are the totally allowed annual and hourly retrieval of commodity c from the stock, respectively.

Environmental commodity limitations

Similar to stock commodities, environmental commodities can also be limited per hour or per year. Both properties are assured by the following two rules:

$$\forall c \in C_{\text{env}}:$$
$$-w \sum_{t \in T_m} \text{CB}(c,t) \leq \overline{M}_c$$

$$\forall c \in C_{\text{env}},\ t \in T_m:$$
$$-\text{CB}(c,t) \leq \overline{m}_c,$$

where $\overline{M}_c$ and $\overline{m}_c$ are the totally allowed annual and hourly emissions of environmental commodity c to the atmosphere, respectively.

Process dispatch constraints

So far, apart from the commodity balance function, the interaction between processes and commodities have not been discussed. It is perhaps in order to start with the general idea behind the modeling of the process operation. In urbs all processes are mimo-processes, i.e., in general they take in multiple commodities as inputs and give out multiple commodities as outputs. The respective ratios between the respective commodity flows remain normally fixed. The operational state of the process is then captured in just one variable, the process throughput τ_{pt} and is is linked to the commodity flows via the following two rules:

$$\forall p \in P,\ c \in C,\ t \in T_m:$$
$$\epsilon^{\text{in}}_{pct} = r^{\text{in}}_{pc} \tau_{pt}$$
$$\epsilon^{\text{out}}_{pct} = r^{\text{out}}_{pc} \tau_{pt},$$

where $r^{\text{in, out}}_{pc}$ are the constant factors linking the commodity flow to the operational state. The efficiency η of the process p for the conversion of commodity c_1 into commodity c_2 is then simply given by:

$$\eta = \frac{r^{\text{out}}_{pc_2}}{r^{\text{in}}_{pc_1}}.$$

Basic process throughput rules

The throughput τ_{pt} of a process is limited by its installed capacity and the specified minimal operational state. Furthermore, the switching speed of a process can be limited:

$$
\begin{aligned}
&\forall p \in P,\ t \in T_m: \\
&\tau_{pt} \leq \kappa_p \\
&\tau_{pt} \geq \underline{P}_p \kappa_p \\
&\tau_{pt} - \tau_{p(t-1)} \leq \Delta t \overline{PG}_p^{\text{up}} \kappa_p \\
&\tau_{pt} - \tau_{p(t-1)} \geq -\Delta t \overline{PG}_p^{\text{down}} \kappa_p
\end{aligned}
,
$$

where $\underline{P}_p$ is the normalized, minimal operational state of the process and $\overline{PG}_p^{\text{up}}$ and $\overline{PG}_p^{\text{down}}$ are the normalized, maximal ramping up gradient, respectively ramping down gradient of the operational state in full capacity per timestep.

Intermittent supply rule

If the input commodity is of type 'SupIm', which means that it represents an operational state rather than a proper material flow, the operational state of the process is governed by this alone. This feature is typically used for renewable energies but can be used whenever a certain operation time series of a given process is desired

$$
\begin{aligned}
&\forall p \in P,\ c \in C_{\text{sup}},\ t \in T_m: \\
&\epsilon_{cpt}^{\text{in}} = s_{ct} \kappa_p.
\end{aligned}
$$

Here, s_{ct} is the time series that governs the exact operation of process p, leaving only its capacity κ_p as a free variable.

This concludes the minimal model.

Intertemporal optimization model

Intertemporal models are a more general type of model than the minimal case. For such models a second time domain is introduced to capture the behavior of the system over a timeframe of many years, thus rendering a modeling of the system development, rather than the optimal system configuration, possible. To keep the model as small as possible while still capturing most of the intertemporal behavior, the second time slice is approximated by a number of support timeframes (years) $Y = (y_1, ..., y_n)$, which is in general smaller than the total model horizon. Each modeled timeframe is then essentially a minimal (or multinode-) model in its own right. The basic approximative assumptions linking the modeled timeframes are then given by:

- Each modeled year is repeated k times if the next modeled year is k years later. The last year is repeated a user specified number of times.
- The depreciation period is assumed to be also the operational period of any unit built. There is no operation in an economically fully depreciated state.
- A unit can only be operated in a given modeled year when it is operational for the entire period that year represents, i.e., until the next modeled year.

- All payments are exponentially discounted with a discount rate j that is set once for the entire modeling horizon.

The variable vector is as a first step only changed in so far, as the second time domain is entering the index. It now reads:

$$x^{\mathrm{T}} = (\zeta, \underbrace{\rho_{yct}}_{\text{commodity variables}}, \underbrace{\kappa_{yp}, \widehat{\kappa}_{yp}, \tau_{ypt}, \epsilon^{\text{in}}_{ycpt}, \epsilon^{\text{out}}_{ycpt}}_{\text{process variables}}).$$

Here, ζ represents the total discounted system costs over the entire modeling horizon, $\rho_y ct$ the amount of commodities c taken from a virtual, infinite stock in year y at time t, κ_{yp} and $\widehat{\kappa}_{yp}$ the total and the newly installed process capacities in year y of processes p, τ_{ypt} the operational state in year y of processes p at time t and $\epsilon^{\text{in}}_{ycpt}$ and $\epsilon^{\text{out}}_{ycpt}$ the total inputs and outputs in year y of commodities c to and from process p at time t, respectively.

All dispatch constraint equations for commodities and processes described in the minimal model section, as well as all such constraints for transmissions, storages, DSM described in their respective dedicated sections, remain structurally the same in an intertemporal model. The only modification there is, that the modeled year shows up as an additional index.

The parts that change in a more meaningful way are the costs and the unit expansion constraints.

Costs

As in the minimal model the total cost variable can be split into the following sum:

$$\zeta = \zeta_{\text{inv}} + \zeta_{\text{fix}} + \zeta_{\text{var}} + \zeta_{\text{fuel}} + \zeta_{\text{env}},$$

where ζ_{inv} are the discounted invest costs accumulated over the entire modeled period, ζ_{fix} the discounted, accumulated fixed costs, ζ_{var} the discounted, sum over the modeled years of all variable costs accumulated over each year, ζ_{fuel} the discounted sum over the modeled years of fuel costs accumulated over each year and ζ_{env} the discounted total penalty for environmental pollution.

All costs are discounted by the same exponent j for the entire modeling horizon on a yearly basis. This means that any payment x that has to be made in a year k will be discounted for the cost function ζ by:

$$x_{\text{discounted}} = (1+j)^{-k} \cdot x$$

Since all non-modeled years are just treated as exact copies of the last modeled year before them, the discounted sum of fix, variable, fuel and environmental costs can simply be taken as the costs of the representative modeled year m multiplied by a factor D_m. If the distance to the next modeled year is k, it can be calculated via:

$$D_m = \sum_{l=m}^{m+k-1} (1+j)^{-l} = (1+j)^{-m} \sum_{l=0}^{k-1} (1+j)^{-l} = (1+j)^{-m} \frac{1-(1+j)^{-k}}{1-(1+j)^{-1}} =$$

$$= (1+j)^{1-m} \frac{1-(1+j)^{-k}}{j}.$$

So for example the variable costs for modeled year m and its k identical, non-modeled copies $\zeta_{\text{var}}^{\{m,m+1,..,m+k-1\}}$ are given by:

$$\zeta_{\text{var}}^{\{m,m+1,..,m+k-1\}} = D_m \cdot \zeta_{\text{var}}^{m},$$

if ζ_{var}^{m} is the sum of all variable costs accumulated by the use of units in the year m alone by the model.

Intertemporal calculation of invest costs

In the intertemporal model, invest costs are calculated using the annuity method. This directly entails that there are no rest values of any units built by the model that have to be considered for the cost function. It is then possible to multiply the annuity payments fC for a unit with investment costs C built in year m simply with the factor D_m. The only difference is, that the investment annuity payments are not restricted to the modeled years but have to be paid for the entire depreciation period provided that it is within the modeled time horizon. When the depreciation period is n and k is the number of payments that fall in the modeled time horizon, the total costs C_{total} of an investment of size C made in year m is given by:

$$C_{\text{m}}^{\text{total}} = D_m \cdot f \cdot C = (1+j)^{1-m} \frac{1-(1+j)^{-k}}{j} \cdot \frac{(1+i)^n \cdot i}{(1+i)^n - 1} \cdot C =$$

$$= \underbrace{(1+j)^{1-m} \cdot \frac{i}{j} \cdot \left(\frac{1+i}{1+j}\right)^n \cdot \frac{(1+j)^n - (1+j)^{n-k}}{(1+i)^n - 1}}_{=:I_{\text{m}}} \cdot C$$

For either $i = 0$ or $j = 0$ a distinction has to be made, which takes the following form:

- $i = 0,\ j = 0$:

$$C_{\text{m}}^{\text{total}} = \underbrace{\frac{k}{n}}_{=:I_{\text{m}}} \cdot C$$

- $i \neq 0,\ j = 0$:

$$C_{\text{m}}^{\text{total}} = k \cdot f \cdot C = \underbrace{k \cdot \frac{(1+i)^n \cdot i}{(1+i)^n - 1}}_{=:I_{\text{m}}} \cdot C$$

- $i = 0,\ j \neq 0$:

$$C_{\text{m}}^{\text{total}} = \frac{1}{n} \cdot (1+j)^{-m} \sum_{l=0}^{k-1} (1+j)^{-l} \cdot C = \underbrace{\frac{1}{n} \cdot (1+j)^{-m} \cdot \frac{(1+j)^k - 1}{(1+j)^k \cdot j}}_{=:I_{\text{m}}} \cdot C$$

In any case the total invest costs are then given by:

$$\zeta_{\text{inv}} = \sum_{y \in Y}$$

$$p \in P C_{\text{m}}^{\text{total}} = \sum_{y \in Y}$$

$$p \in P I_{\text{y}} k_{yp}^{\text{inv}} \widehat{\kappa}_{yp}$$

Unit expansion constraints

Apart from the costs there are also changes in the unit expansion constraints for an intertemporal model. These changes mostly concern how the amount of installed units is found. This becomes an issue since

units built in an earlier modeled year or already installed in the first modeled year, may or may not be operational in a given modeled year m and through $m + k - 1$. Here, k is the distance to the next modeled year or the end of the modeled horizon in case of m being the last modeled year. To properly calculate the capacity of a process in a year y the following two sets are necessary:

$$O := \{(p, y_i, y_j) | p \in P,\ \{y_i, y_j\} \in Y,\ y_i \leq y_j,\ y_i + L_p \geq\ y_{j+1}\}$$

$$O_{\text{inst}} := \{(p, y_j) | p \in P_0,\ y \in Y,\ y_0 + T_p \geq y_{j+1}\},$$

where L_p is the lifetime of processes p, P_0 the subset of processes that are already installed in the first modeled year y_0 and T_p the rest lifetime of already installed processes. If y_j is the last modeled year, y_{j+1} stands for the end of the model horizon.

With these two sets the installed process capacity in a given year is then given by:

$$\kappa_{yp} = \sum_{y' \in Y} \\ (p, y', y) \in O \widehat{\kappa}_{y'p} + K_p\ ,\ \text{ if } (p, y) \in O_{\text{inst}}$$

$$\kappa_{yp} = \sum_{y' \in Y} \\ (p, y', y) \in O \widehat{\kappa}_{y'p}\ ,\ \text{ else}$$

where K_p is the installed capacity of process p at the beginning of the modeling horizon. Since for each modeled year still the capacity constraint

$$\forall y \in Y,\ p \in P: \\ \underline{K}_{yp} \leq \kappa_{yp} \leq \overline{K}_{yp}$$

is valid, the set constraints can have effects across years and especially the modeller has to be careful not to set infeasible constraints.

Commodity dispatch constraints

While in an intertemporal model all commodity constraints within one modeled year remain valid one addition is possible concerning CO2 emissions. Here, a budget can be given, which is valid over the entire modeling horizon:

$$-w \sum_{y \in Y} \\ t \in T_m \text{CB}(y, \text{CO}_2, t) \leq \overline{\overline{L}}_{\text{CO}_2}$$

Here, $\overline{\overline{L}}_c$ is the global budget for the emission of the environmental commodity. Currently this is hard coded for CO2 only.

This rule concludes the model additions introduced by intertemporal modeling.

Multinode optimization model

The introduction of multiple spatial nodes into the model is the second big extension of the minimal model that is possible. Similar to the intertemporal model expansion it also adds an index level to

all variables and parameters. This addition is perpendicular to the intertemporal modeling and both extensions do not interact in any complex way with each other. Here, the multinode model extension will be shown without the intertemporal extension, i.e., it is shown as an extension to the minimal model. In this case the variable vector of the optimization problem reads:

$$x^{\mathrm{T}} = (\zeta, \underbrace{\rho_{vct}}_{\text{commodity variables}}, \underbrace{\kappa_{vp}, \widehat{\kappa}_{vp}, \tau_{vpt}, \epsilon^{\text{in}}_{vcpt}, \epsilon^{\text{out}}_{vcpt}}_{\text{process variables}}, \underbrace{\kappa_{af}, \widehat{\kappa}_{af}, \pi^{\text{in}}_{aft}, \pi^{\text{out}}_{aft}}_{\text{transmission variables}}).$$

Here, ζ represents the total annualized system cost across all modeled vertices $v \in V$, ρ_{vct} the amount of commodities c taken from a virtual, infinite stock at vertex v and time t, κ_{vp} and $\widehat{\kappa}_{vp}$ the total and the newly installed process capacities of processes p at vertex v, τ_{vpt} the operational state of processes p at vertex v and time t, $\epsilon^{\text{in}}_{vcpt}$ and $\epsilon^{\text{out}}_{vcpt}$ the total inputs and outputs of commodities c to and from process p at vertex v and time t, κ_{af} and $\widehat{\kappa}_{af}$ the installed and new capacities of a transmission line f linking two vertices with the arc a and π^{in}_{aft} and π^{out}_{aft} the in- and outflows into arc a via transmission line f at time t.

There are no qualitative changes to the cost function only the sum of all units now extends over processes and transmission lines.

Transmission capacity constraints

Transmission lines are modeled as unidirectional arcs in urbs. This means that they have a input site and an output site. Furthermore, an arc already specifies a commodity that can travel across it. However, from the modelers point of view the transmissions rather behave like non-directional edges, linking both sites with the identical capacity in both directions. This behavior is then ensured by the transmission symmetry rule, which sets the capacity of both unidirectional arcs to be identical:

$$\forall a \in V \times V \times C,\ f \in F:$$
$$\kappa_{af} = \kappa_{a'f},$$

where a' is the inverse arc of a. The transmission capacity is then calculated similar to process capacities in the minimal model:

$$\forall a \in V \times V \times C,\ f \in F:$$
$$\kappa_{af} = K_{af} + \widehat{\kappa}_{af},$$

where K_{af} represents the already installed and $\widehat{\kappa}_{af}$ the new capacity of transmission f installed in arc a. The new capacity can also be expressed as the product of the parameter transmission capacity block K^{block}_{yaf} and the variable new transmission capacity units β_{yaf}:

$$\widehat{\kappa}_{af} = K^{\text{block}}_{yaf} \cdot \beta_{yaf}$$

Transmission capacity limit rule

Completely analogous to processes also transmission line capacities are limited by a maximal and minimal allowed capacity $\overline{K}_{af}$ and $\underline{K}_{af}$ via:

$$\forall a \in V \times V \times C,\ f \in F:$$
$$\underline{K}_{af} \leq \kappa_{af} \leq \overline{K}_{af}$$

Commodity dispatch constraints

Apart from these time independent rules, the time dependent rules governing the unit utilization are amended with respect to the minimal model by the introduction of transmission lines.

Amendments to the Vertex rule

The vertex rule is changed, since additional commodity flows through the transmission lines occur in each vertex. The commodity balance function is thus changed to:

$$\forall c \in C,\ t \in T_m :$$

$$\mathrm{CB}(c,t) = \sum_{(c,p)\in C_p^{\mathrm{in}}} \epsilon_{vcpt}^{\mathrm{in}} + \sum_{(a,f)\in A_v^{\mathrm{in}}} \pi_{aft}^{\mathrm{in}} - \sum_{(c,p)\in C_p^{\mathrm{out}}} \epsilon_{vcpt}^{\mathrm{out}} - \sum_{(a,f)\in A_v^{\mathrm{out}}} \pi_{aft}^{\mathrm{out}}.$$

Here, the new tuple sets $A_v^{\mathrm{in,out}}$ represent all input and output arcs a connecting vertex v, respectively. The commodity balance is thereby allowing for commodities to leave the system at vertex v via arcs as well as processes. Apart from this change to the commodity balance the vertex rule and the other rules restricting commodity flows remain unchanged with respect to the minimal model.

Global CO2 limit

In addition to the general vertex specific constraint for the emissions of environmental commodities as discussed in the minimal model, there is a hard coded possibility to limit the CO2 emissions across all modeled sites:

$$-w \sum_{v\in V}$$
$$t \in T_m \mathrm{CB}(v, \mathrm{CO_2}, t) \leq \overline{L}_{\mathrm{CO_2},y}$$

Transmission dispatch constraints

There are two main constraints for the commodity flows to and from transmission lines. The first restricts the total amount of commodity c flowing in arc a on transmission line f to the total capacity of the line:

$$\forall a \in V \times V \times C,\ f \in F,\ t \in T_m :$$
$$\pi_{aft}^{\mathrm{in}} \leq \kappa_{af}.$$

Here, the input into the arc π_{aft}^{in} is taken as a reference for the total capacity. The output of the arc in the target site is then linked to the input with the transmission efficiency e_{af}

$$\forall a \in V \times V \times C,\ f \in F,\ t \in T_m :$$
$$\pi_{aft}^{\mathrm{out}} = e_{af} \cdot \pi_{aft}^{\mathrm{in}}.$$

DC Power Flow feature

Transmission lines can be modelled with DC Power Flow as an optional feature to represent the AC network grid. With the DC Power Flow feature, the variable voltage angle is introduced for the vertices connected with DC Power Flow transmission lines The DC Power Flow is defined by the relation between the voltage angle θ_{vt} of connecting vertices.

It is possible to combine the default transmission model with the DC Power Flow transmission model. The DCPF feature can be activated on the selected transmission lines. This way two different sets of transmission tuples, subject to different constraints, will be modelled. These transmission tuple sets are defined as the set of transport model (default) transmission lines $F_{cv_{\mathrm{out}}v_{\mathrm{in}}}^{TP}$ and the set of DCPF transmission lines $F_{cv_{\mathrm{out}}v_{\mathrm{in}}}^{DCPF}$

Usage

This feature can be activated for selected transmission lines by including the following parameters:

- The reactance X_{af} of a transmission line is required to be included in the model input to model the given transmission line with DCPF. This parameter should be greater than 0 and given in per-unit system. If this parameter is excluded from the model input, DCPF will not be activated for the transmission line.
- The voltage angle difference of two connecting sites should be limited with angle difference limit $\overline{dl}_{af}$ to create a stable model. This parameter is required to limit the voltage angle difference between two connecting sites. A degree value between 0 and 91 is allowed.
- The base voltage $V_{af\text{base}}$ of transmission lines are required to convert the power flow from per-unit system to MW. The base voltage parameter is required in kV for every transmission line, which should be modelled with DCPF. The value of this parameter should be greater than 0.
- Since the DC Power Flow model ignores the loss of a transmission line, the efficiency e_{af} of the transmission lines modelled with the DCPF should be set to 100% represented with the value "1".

Contrary to the default transmission line representation, DC Power Flow transmission lines are represented with a single bidirectional arc between two vertices. The complementary arc of a DC Power Flow transmission line will be excluded from the model even if it is defined by the user. Depending on the voltage angle difference of two connecting sites, the power flow π_{aft} on a DC Power Flow transmission line can be both negative or positive indicating the direction of the flow.

DC Power Flow Equation

Power flow on a transmission line modelled with DCPF:

$$\pi^{\text{in}}_{aft} = \frac{(\theta_{v_{\text{in}}t} - \theta_{v_{\text{out}}t})}{57.2958}(-\frac{-1}{X_{af}})V^2_{af\text{base}}$$

Here $\theta_{v_{\text{in}}t}$ and $\theta_{v_{\text{out}}t}$ are the voltage angles of the source site v_{in} and destinaton site v_{out}. These are converted to radian from degrees by dividing by 57,2958. X_{af} is the reactance of the transmission line in per unit system and $(-\frac{-1}{X_{af}})$ is the admittance of the transmission line.

Constraints

Constraints applied to the DCPF transmission lines vary from those applied to the transport transmission lines.

Symmetry rule is ignored for the DCPF transmission lines, since these lines only consist of single bidirectional arcs. Since the DCPF transmission lines do not have complementary arcs the fixed and investment costs would be halved. To prevent this error caused by the excluded symmetry constraint for DCPF transmission lines, fixed and investment prices for DCPF lines are doubled automatically before calculating the costs.

The constraint which restricts the commodity flow π^{in}_{aft} on a transmission line with the installed capacity κ_{af} is expanded for DCPF transmission lines. The additional constraint restricts the lower limit of the power flow, since the power flow with DCPF can also be negative.

$$-\pi^{\text{in}}_{aft} \leq \kappa_{af}$$

Voltage angle difference of two connecting vertices v_{in} and v_{out} is restricted with the angle difference limit parameter $\overline{dl}_{af}$ given for a DCPF transmission f on an arc a

$$-\overline{dl}_{af} \leq (\theta_{v_{\text{in}}t} - \theta_{v_{\text{out}}t}) \leq \overline{dl}_{af}$$

Two additional constraints are used in DCPF feature to retrieve the absolute value ${\pi^{\text{in}}_{aft}}'$ of the power flow on a DCPF transmission line, which is included in the variable cost calculation. With the help of these constraints and minimization of objective function , which includes the substitute variable ${\pi^{\text{in}}_{aft}}'$, the substitute variable will be equal to the absolute value of the power flow variable $|\pi^{\text{in}}_{aft}|$

$${\pi^{\text{in}}_{aft}}' \geq \pi^{\text{in}}_{aft}$$

$${\pi^{\text{in}}_{aft}}' \geq -\pi^{\text{in}}_{aft}$$

Energy Storage

Storages can optionally be set in urbs. They introduce additional variables and constraints, contribute to the cost function but do not increase the index depth of all variables and parameters. For this and all the further features all variables will be written in the full index depth, i.e. for intertemporal models with multiple vertices. For storages the capacity and the charging/discharging power are expanded independently. For each storage one commodity is specified which is stored. It is thus not necessary to specify the commodity as an extra index in the variables and parameters. With added storages the variable vector then reads:

$$x^{\text{T}} = (\zeta, \underbrace{\rho_{yvct}}_{\text{commodity variables}}, \underbrace{\kappa_{yvp}, \widehat{\kappa}_{yvp}, \tau_{yvpt}, \epsilon^{\text{in}}_{yvcpt}, \epsilon^{\text{out}}_{yvcpt}}_{\text{process variables}}, \underbrace{\kappa_{yaf}, \widehat{\kappa}_{yaf}, \pi^{\text{in}}_{yaft}, \pi^{\text{out}}_{yaft}}_{\text{transmission variables}},$$

$$\underbrace{\kappa^{\text{c}}_{yvs}, \kappa^{\text{p}}_{yvs}, \widehat{\kappa}^{\text{c}}_{yvs}, \widehat{\kappa}^{\text{p}}_{yvs}, \epsilon^{\text{in}}_{yvst}, \epsilon^{\text{out}}_{yvst}, \epsilon^{\text{con}}_{yvst}}_{\text{storage variables}}).$$

Here, the new storage variables $\kappa^{\text{c,p}}_{yvs}$ and $\widehat{\kappa}^{\text{c,p}}_{yvs}$ stand for the total and new capacities of storage capacity and power of storage unit s, in modeled year y at vertex v, respectively, the variables $\epsilon^{\text{in,out}}_{yvst}$ represent the input and output to storage s in year y at vertex v at time t and $\epsilon^{\text{con}}_{yvst}$ the storage state.

Costs

The costs are changed in a straightforward way. The invest, fix and variable costs are now summed over the storage capacities, powers and the total amount of charged and discharged commodity in addition to the process indices. As in the case of transmissions there are no qualitative changes to the costs.

Storage expansion constraints

Storages are expanded in their capacity and charging and discharging power separately. The respective constraints read:

$$\kappa^{\text{c,p}}_{yvs} = \sum_{y' \in Y} (s, v, y', y) \in O\widehat{\kappa}^{\text{c,p}}_{y'vs} + K_{vs}\ , \ \text{ if } (s, v, y) \in O_{\text{inst}}$$

$$\kappa^{\text{c,p}}_{yvs} = \sum_{y' \in Y} (s, v, y', y) \in O\widehat{\kappa}^{\text{c,p}}_{y'vs}\ , \ \text{ else,}$$

where $\kappa^{\text{c,p}}_{yvs}$ are the total installed capacity and power, repectively, in year y at site v of storage s and $\widehat{\kappa}^{\text{c,p}}_{yvs}$ the corresponding

newly installed storage capacities and powers. Both newly installed quantities can also be expressed as the product of the parameter storage new capacity/power block $K^{\text{c,p block}}_{yvs}$ and the variable new storage size/power units $\beta^{\text{c,p}}_{yvs}$:

$$\widehat{\kappa}^{\text{c,p}}_{yvs} = K^{\text{c,p block}}_{yvs} \cdot \beta^{\text{c,p}}_{yvs}$$

Both total installed quantities are then also given an upper and a lower bond via:

$$\begin{aligned}
&\forall y \in Y,\ v \in V,\ s \in S: \\
&\underline{K}^{\text{c}}_{yvs} \leq \kappa^{\text{c}}_{yvs} \leq \overline{K}^{\text{c}}_{yvs} \\
&\underline{K}^{\text{p}}_{yvs} \leq \kappa^{\text{p}}_{yvs} \leq \overline{K}^{\text{p}}_{yvs}
\end{aligned}$$

Commodity dispatch constraints

The commodity unit utilization constraints are expanded by the use of storages.

Amendments to the Vertex rule

The vertex rule is changed, since additional commodity flows into and out of the storages can occur. The commodity balance function is thus changed to:

$$\forall y \in Y,\ v \in V,\ c \in C,\ t \in T_m:$$

$$\begin{aligned}
\text{CB}(y, v, c, t) = &\sum_{(y,v,c,p) \in C^{\text{in}}_{y,v,c,p}} \epsilon^{\text{in}}_{vcpt} + \sum_{(y,v,s,c) \in C_{y,v,s,c}} \epsilon^{\text{in}}_{yvst} + \sum_{(y,a,f) \in A^{\text{in}}_{v}} \pi^{\text{in}}_{aft} - \\
&- \sum_{(y,v,c,p) \in C^{\text{out}}_{p}} \epsilon^{\text{out}}_{vcpt} - \sum_{(y,v,s,c) \in C_{y,v,s,c}} \epsilon^{\text{out}}_{yvst} - \sum_{(y,a,f) \in A^{\text{out}}_{v}} \pi^{\text{out}}_{aft}.
\end{aligned}$$

Here, the new tuple sets $C^{\text{in,out}}_{y,v,s,c}$ represent all inputs and outputs in year y at vertex v of commodity c into storage s. The variables $\epsilon^{\text{in,out}}_{yvst}$ are then the inputs and outputs of commodities to and from storages.

Storage dispatch constraints

In a storage the energy content $\epsilon^{\text{con}}_{yvst}$ has to be calculated. This is achieved by simply adding all inputs to and subtracting all outputs from the storage content at the previous time step $\epsilon^{\text{con}}_{yvs(t-1)}$:

$$\forall y \in Y,\ v \in V,\ s \in S,\ t \in T_m:$$
$$\epsilon^{\text{con}}_{yvst} = \epsilon^{\text{con}}_{yvs(t-1)} \cdot (1 - d_{yvs})^{\Delta t} + e^{\text{in}}_{yvs} \cdot \epsilon^{\text{in}}_{yvst} - \frac{\epsilon^{\text{out}}_{yvst}}{e^{\text{out}}_{yvs}}.$$

Here, $e^{\text{in,out}}_{yvs}$ are the efficiencies for charging and discharging, respectively, and d_{yvs} is the hourly self discharge rate.

Basic storage dispatch rules

Similar to processes and transmission lines, inputs and outputs are limited by the power capacity of the storage:

$$\forall y \in Y,\ v \in V,\ s \in S,\ t \in T_m:$$
$$\epsilon^{\text{in,out}}_{yvst} \leq \Delta t \cdot \kappa^{\text{p}}_{yvs}.$$

Additionally, the storage content is limited by the total storage energy capacity:

$$\forall y \in Y,\ v \in V,\ s \in S,\ t \in T_m:$$
$$\epsilon^{\text{con}}_{yvst} \leq \kappa^{\text{c}}_{yvs}.$$

Initial and final state

In order to avoid windfall profits for the optimization by, e.g., emptying a storage over the model horizon, the initial and final storage content are linked via:

$$\forall y \in Y,\ v \in V,\ s \in S:$$
$$\epsilon^{\text{con}}_{yvs(t_1)} \leq \epsilon^{\text{con}}_{yvst_N},$$

where $t_{1,N}$ are the initial and final modeled timesteps, respectively. The inequality simplifies the model solving by relaying an otherwise unnecessarily strict constraint. A small disadvantage arises when the system can gain costs or save CO2 by filling a storage. This case is, however, not too common. It is additionally possible for the user to fix the initial storage content via:

$$\forall y \in Y,\ v \in V,\ s \in S:$$
$$\epsilon^{\text{con}}_{vst_1} = \kappa^{\text{c}}_{yvs} I_{yvs},$$

where I_{yvs} is the fraction of the total storage capacity that is filled at the beginning of the modeling period.

Fixed energy/power ratio

It is sometimes desirable to fix the ratio between energy capacity and charging/discharging power for a given storage. This is modeled by the possibility to set a linear dependence between the capacities through a user-defined "energy to power ratio" $k^{\text{E/P}}_{yvs}$. Note that this constraint is only active for the

storages with a positive value under the column "ep-ratio" in the input file, and when this value is not given, the power and energy capacities can be sized independently

$$\forall y \in Y,\ v \in V,\ s \in S:$$
$$\kappa^{c}_{yvs} = \kappa^{p}_{yvs} k^{\text{E/P}}_{yvs}.$$

This concludes the storage feature.

Trading with an external market

In urbs it is possible to model the trade with an external market. For this two new commodity types, buy and sell commodities, are introduced. For each a time series representing the momentary cost at each timestep is given. This time series is of course known to the model in advance, which has two implications. First, the modeled system is considered too small to influence the external market and any possible influence is not captured by the model, and, second, the perfect price foresight can distort the results when compared to a realistic trader in a market. For models with buy and sell commodities the variable vector takes the following form:

$$x^{\mathrm{T}} = (\zeta, \underbrace{\rho_{yvct}, \varrho_{yvct}, \psi_{yvct}}_{\text{commodity variables}}, \underbrace{\kappa_{yvp}, \widehat{\kappa}_{yvp}, \tau_{yvpt}, \epsilon^{\text{in}}_{yvcpt}, \epsilon^{\text{out}}_{yvcpt}}_{\text{process variables}}, \underbrace{\kappa_{yaf}, \widehat{\kappa}_{yaf}, \pi^{\text{in}}_{yaft}, \pi^{\text{out}}_{yaft}}_{\text{transmission variables}}),$$

where ϱ_{yvct} is the amount of sell commodity c sold to the external market in year y from vertex v at time t and ψ_{yvct} is the amount of buy commodity c bought from the external market in year y at vertex v and time t.

Costs

The cost function is amended by two new cost types when the trading with an external market is modeled, the purchase and the revenue costs

$$\zeta = \zeta_{\text{inv}} + \zeta_{\text{fix}} + \zeta_{\text{var}} + \zeta_{\text{fuel}} + \zeta_{\text{rev}} + \zeta_{\text{pur}} + \zeta_{\text{env}}.$$

The two new cost types are then specified by the following equations:

$$\begin{aligned} \zeta_{\text{rev}} = & - w \Delta t \sum_{y \in Y} \\ & v \in V \\ & c \in C_{sell} \\ & t \in T_m D_m \cdot k^{\text{bs}}_{yvct} \cdot \varrho_{yvct} \end{aligned}$$

$$\begin{aligned} \zeta_{\text{pur}} = & w \Delta t \sum_{y \in Y} \\ & v \in V \\ & c \in C_{buy} \\ & t \in T_m D_m \cdot k^{\text{bs}}_{yvct} \cdot \psi_{yvct}, \end{aligned}$$

where k^{bs}_{yvct} represents the time series of the given buy and sell commodity prices.

Commodity dispatch constraints

Buy and sell commodities change the vertex rule (Kirchhoff's current law), by adding a new way for in- an output flows of commodities. The rule is thus amended by the following two equations:

$$\forall y \in Y,\ v \in V,\ c \in C_{\text{sell}},\ t \in T_m :$$
$$-\varrho_{ct} \geq \text{CB}(c,t)$$

$$\forall y \in Y,\ v \in V,\ c \in C_{\text{buy}},\ t \in T_m :$$
$$\psi_{ct} \geq \text{CB}(c,t).$$

The commodity balance itself is not changed. The new rules state that any amount of energy sold needs to be provided to (negative CB) the system via processes, storages or transmission lines, while buy commodity consumed by processes, storages or transmission lines in the system has to be replenished.

Buy/sell commodity limitations

The trade with the market in each modeled year and each vertex can be limited per time step and for an entire year. This introduces the following constraints:

$$\forall y \in Y,\ v \in V,\ c \in C_{\text{sell}} :$$
$$w \sum_{t \in T_m} \varrho_{ct} \leq \overline{G}_{yvc}$$

$$\forall y \in Y,\ v \in V,\ c \in C_{\text{sell}},\ t \in T_m :$$
$$\varrho_{yvct} \leq \overline{g}_{yvc}$$

and

$$\forall y \in Y,\ v \in V,\ c \in C_{\text{buy}} :$$
$$w \sum_{t \in T_m} \psi_{ct} \leq \overline{B}_{yvc}$$

$$\forall y \in Y,\ v \in V,\ c \in C_{\text{buy}},\ t \in T_m :$$
$$\varrho_{yvct} \leq \overline{b}_{yvc}.$$

Here, the parameters $\overline{b}_{yvc}$ and $\overline{B}_{yvc}$ limit the hourly and yearly maximums of buy from and $\overline{g}_{yvc}$ and $\overline{G}_{yvc}$ the hourly and yearly maximum of selling to the external market.

This concludes the discussion of the modeled trading with an external market.

Demand side management

Demand side management allows for the shifting of demands in time. It thus gives the model the possibility to divert from the strict restriction that all demands have to be fulfilled at all timesteps. Demand side management adds two variables to an urbs problem and the variable vector then reads:

$$x^{\text{T}} = (\zeta, \underbrace{\rho_{yvct}}_{\text{commodity variables}}, \underbrace{\kappa_{yvp}, \widehat{\kappa}_{yvp}, \tau_{yvpt}, \epsilon^{\text{in}}_{yvcpt}, \epsilon^{\text{out}}_{yvcpt}}_{\text{process variables}}, \underbrace{\kappa_{yaf}, \widehat{\kappa}_{yaf}, \pi^{\text{in}}_{yaft}, \pi^{\text{out}}_{yaft}}_{\text{transmission variables}}, \underbrace{\delta^{\text{up}}_{yvct}, \delta^{\text{down}}_{yvct(tt)}}_{\text{DSM variables}}).$$

The new variable $\delta^{\text{up}}_{yvct}$ represent the upshift of the momentary demand at time t and $\delta^{\text{down}}_{yvct(tt)}$ the corresponding downshifts. The downshifts need two time indices as they are referencing to the corresponding upshift with the first index t and the timesteps they actually occur via the second time index tt. The latter is then restricted to an interval around the reference upshift since loads cannot in general be shifted indefinitely. As it is modeled in urbs, DSM does not introduce any costs. To clarify the terms used for the DSM feature the following illustrative example is helpful.

Example of a DSM process

An example scenario with parameters below can be used to clarify the mathematical structure of a DSM process.

Site	Commodity	delay	eff	recov	cap-max-do	cap-max-up
South	Elec	3	1	1	2000	2000

First, an series of three upshifts, i.e. demand increases, indexed with the modeled timesteps 3,4 and 5 occurs in the example.

Table 1: **DSM upshift process**

t	
1	0
2	0
3	1445
4	1580
5	2000
6	0

The corresponding downshifts can then be visualized using a matrix, where the row index t corresponds to the upshifts above, that have to be compensated by downshifts. The modeled timesteps where the downshifts actually occur are labeled by tt and represent the column indices.

Table 2: **DSM downshift process**

$t \setminus tt$	1	2	3	4	5	6
1	0	0	0	0		
2	0	0	0	0	0	
3	1445	0	0	0	0	0
4	555	0	555	0	0	470
5		2000	0	0	0	0
6			0	0	0	0

The DSM upshift process is relatively easy to understand, for every time step t one upshift is made and it can not exceed 2000. The table for DSM downshift process shows, that the sum over all elements for every row index t, is equal to the upshift made at time step t. The blank spaces in the table are because of delay time restriction. For instance, an upshift in $t = 1$ may not be compensated with a downshift in $tt = 5$, as delay time is equal to 3 in our example. The restriction of the total DSM downshifts is given by the sum of all column elements for every index tt. This sum may not exceed 2000 as well, due to given parameters.

Commodity dispatch constraints

Demand side management changes the vertex rule. Every upshift $\delta^{\text{up}}_{yvct}$ leads to an additional demand, i.e., to an additional required output of the system, and vice versa for the downshifts. Effectively this changes the vertex rule (Kirchhoff's current law) for demand commodities with DSM to:

$$\forall y \in Y,\ v \in V,\ c \in C_{\text{dem}},\ t \in T_m:$$

$$-d_{yvct} - \delta^{\text{up}}_{yvct} \geq \text{CB}(y,v,c,t)$$
$$-d_{yvct} + \sum_{tt \in [t-y_{yvc}, t+y_{yvc}]} \delta^{\text{down}}_{yvc(tt)t} \geq \text{CB}(y,v,c,t).$$

The downshift equation requires a little elaboration. Here, the total downshift occurring at a timestep t can be caused by downshifts linked to different upshifts, which in the notation above occur at times tt. All downshift contributions within the delay time y_{yvc} of their respective upshifts are then summed up.

DSM variables rule

This central constraint rule for DSM in urbs links the up- and down shifts of DSM events. An upshift (multiplied with the DSM efficiency) at time t must be compensated with multiple downshifts during a certain time interval. The lower and upper bounds of this time interval are given by $t - y_{yvc}$ and $t + y_{yvc}$, where y_{yvc} is the delay time parameter specifying the allowed duration of a DSM event. Inside this time interval, another time index tt is required. It is used to index the downshift processes that are always linked to one upshift. Of course, the intervals of several upshifts can overlap. Mathematically, this rule can be noted like this:

$$\forall y \in Y,\ v \in V,\ c \in C^{\text{DSM}}_{dem},\ t \in T_m:$$

$$e_{yvc}\delta^{\text{up}}_{yvct} = \sum_{tt \in [t-y_{yvc}, t+y_{yvc}]} \delta^{\text{down}}_{yvct(tt)},$$

where e_{yvc} is the DSM efficiency. Note here, that the summation is over the timesteps where the downshifts are occurring as opposed to the vertex rule above, where the summation is over the timesteps of the corresponding upshifts.

DSM shift limitations

DSM shifts are limited in size in both directions. This is modeled by

$$\forall y \in Y,\ v \in V,\ c \in C^{\text{DSM}}_{\text{dem}}, t \in T_m:$$

$$\delta^{\text{up}}_{yvct} \leq \overline{K}^{\text{up}}_{yvc}$$

$$\sum_{tt \in [t-y_{yvc}, t+y_{yvc}]} \delta^{\text{down}}_{yvc(tt)t} \leq \overline{K}^{\text{down}}_{yvc},$$

where again the downshifts are summed over the corresponding upshifts, making sure that at no time there is a total downshift sum larger than the set maximum value.

In addition to these limitations on the single shift directions, the total sum of shifts is also limited in an urbs model via:

$$\forall y \in Y,\ v \in V,\ c \in C^{\text{DSM}}_{\text{dem}}, t \in T_m :$$

$$\delta^{\text{up}}_{yvct} + \sum_{tt \in [t-y_{yvc}, t+y_{yvc}]} \delta^{\text{down}}_{yvc(tt)t} \leq \max\{\overline{K}^{\text{up}}_{yvc}, \overline{K}^{\text{down}}_{yvc}\}.$$

DSM recovery

Assuming that DSM is linked to some real physical devices, it is necessary to allow these devices to have some minimal time between DSM events, where, e.g., the ability to perform DSM is recovered. This is modeled in the following way:

$$\forall y \in Y,\ v \in V,\ c \in C^{\text{DSM}}_{\text{dem}}, t \in T_m :$$

$$\sum_{tt=t}^{o_{yvc}/\Delta t - 1} \delta^{\text{up}}_{yvc(tt)} \leq \overline{K}^{\text{up}}_{yvc} \cdot y_{yvc},$$

where o_{yvc} is the recovery time in hours. This constraint limits the total amount of upshifted energy within the recovery period (lhs) to the maximum allowed energy shift retained for the maximum amount of allowed shifting time for one shifting event. This means that only one full shifting event can occur within the recovery period.

This concludes the demand side management constraints.

Advanced Processes

Several processes have a complicated, non-linear behavior. Those that can be modelled in urbs are explained here. These are: Time Variable Efficiency, Minimum Load and Part Load Behaviors and On/Off Behavior.

Time Variable Efficiency

It is possible to exogenously manipulate the output of a process by introducing a time series, which changes the output ratios and thus the efficiency of a given process in each given timestep. This introduces an additional set of constraints in the form:

$$\forall p \in P^{\text{TimeVarEff}},\ c \in C - C^{\text{env}}, t \in T_m :$$

$$\epsilon^{\text{out}}_{ypct} = r^{\text{out}}_{ypc} f^{\text{out}}_{ypt} \tau_{ypct}.$$

Here, f^{out}_{pt} represents the normalized time series of the varying output ratio. This feature can be helpful when modeling, e.g., temperature dependent effects or maintenance intervals. Environmental commodities are intentionally excluded from the output manipulation. The reason for this is that they are typically directly linked to inputs as, e.g., CO2 emissions are linked to the fossil inputs. A manipulation of the output for environmental commodities would thus violate the mass balance of carbon in this case (e.g. coal).

When the process in question is a process with part load behavior the equation for the time variable efficiency case takes the following form:

$$\forall p \in P^{\text{part load}} \text{ and } p \in P^{\text{TimeVarEff}},\ c \in C,\ t \in T_m:$$

$$\epsilon^{\text{out}}_{ypct} = \Delta t \cdot f^{\text{out}}_{ypt} \cdot \left(\frac{\underline{r}^{\text{out}}_{ypc} - r^{\text{out}}_{ypc}}{1 - \underline{P}_{yp}} \cdot \underline{P}_{yp} \cdot \kappa_{yp} + \frac{r^{\text{out}}_{ypc} - \underline{P}_p \underline{r}^{\text{out}}_{ypc}}{1 - \underline{P}_{yp}} \cdot \tau_{ypt} \right).$$

Minimum Load and Part Load Behaviors

There are some processes which theoretically can be turned on and off, while others tipically operate as must-run units (e.g. nuclear power plants, heat-producing plants during the cold season etc.). These processes can either have a constant and load independent efficiency or a part-load behavior.

In the case of a minimum load behavior with a constant, load independent efficiency, the values of the input and of the output of a process remain unchanged when compared except for the fact that their values, together with the value of the throughput, stay between the following boundaries:

$$\forall p \in P^{\text{minimum load}},\ c \in C,\ t \in T_m:$$

$$\underline{P}_p \cdot \kappa_p \cdot r^{\text{in,out}} \leq \epsilon^{\text{in,out}}_{pct} \leq \kappa_p \cdot r^{\text{in,out}},$$

$$\forall p \in P^{\text{part load}},\ c \in C,\ t \in T_m:$$

$$\underline{P}_p \cdot \kappa_p \cdot \underline{r}^{\text{in,out}} \leq \epsilon^{\text{in,out}}_{pct} \leq \kappa_p \cdot r^{\text{in,out}},$$

where $\underline{P}_p$ is the minimum load fraction, κ_p the installed capacity, $r^{\text{in,out}}$ the input/output ratios and $\underline{r}^{\text{in,out}}$ the minimum input/output ratios.

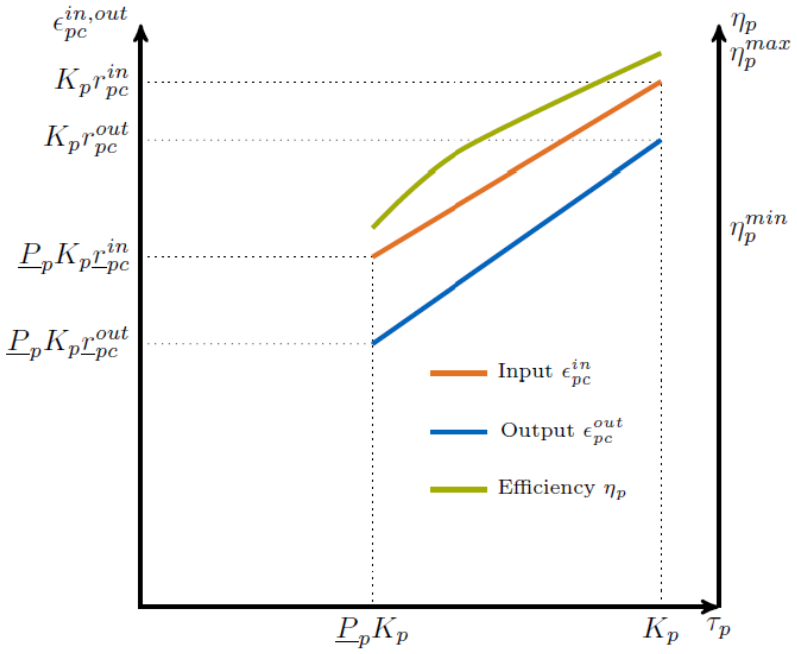
Many processes show a non-trivial part-load behavior. In particular, often a nonlinear reaction of the efficiency on the operational state is given. Although urbs itself is a linear program this can with some caveats be captured in many cases. The reason for this is, that the efficiency of a process is itself not given as a parameter, but is merely the ratio between input and output multipliers. It is thus possible to use purely linear functions to get a nonlinear behavior of the efficiency of the form:

$$\eta = \frac{a + b\tau_{pt}}{c + d\tau_{pt}},$$

where a,b,c and d are some constants. Specifically, the input and output ratios can be set to vary linearly between their respective values at full load $r^{\text{in,out}}_{pc}$ and their values at the minimal allowed operational state $\underline{P}_p \kappa_p$, which are given by $\underline{r}^{\text{in,out}}_{pc}$. This is achieved with the following equations and exemplified with the following graphic:

$$\forall p \in P^{\text{part load}},\ c \in C,\ t \in T_m:$$

$$\epsilon^{\text{in,out}}_{pct} = \Delta t \cdot \left(\frac{\underline{r}^{\text{in,out}}_{pc} - r^{\text{in,out}}_{pc}}{1 - \underline{P}_p} \cdot \underline{P}_p \cdot \kappa_p + \frac{r^{\text{in,out}}_{pc} - \underline{P}_p \underline{r}^{\text{in,out}}_{pc}}{1 - \underline{P}_p} \cdot \tau_{pt} \right).$$

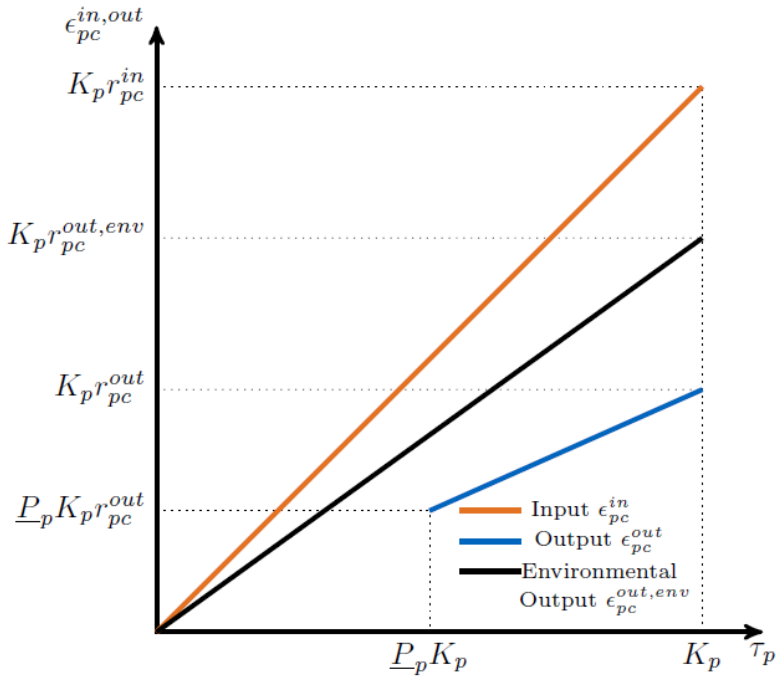


A few restrictions have to be kept in mind when using this feature:

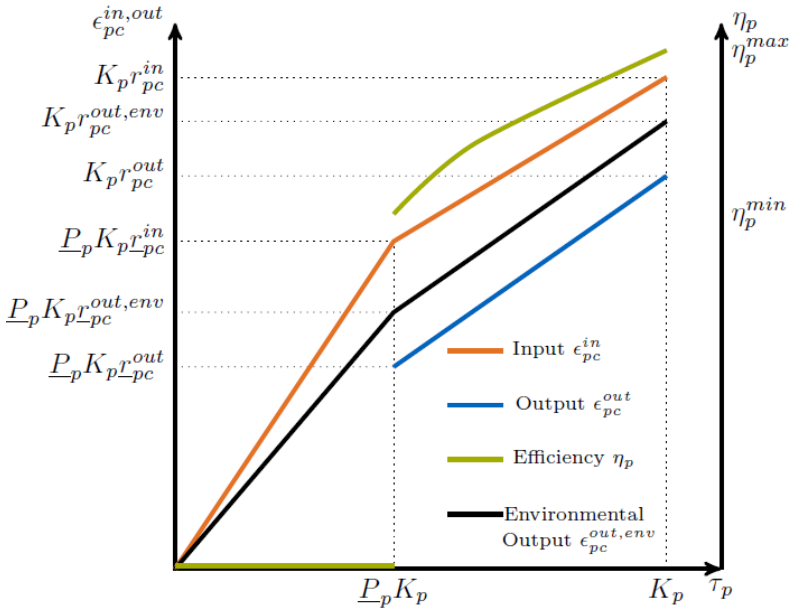
- $\underline{P}_p$ has to be set larger than 0 otherwise the feature will work but not have any effect.
- Environmental output commodities have to mimic the behavior of the inputs by which they are generated. Otherwise the emissions per unit of input would change together with the efficiency, which is typically not the desired behavior.

On/off Behavior

Some processes are characterised by a minimum or part-load behavior but still retain the practical necessity of being turned on and off if this is optimal. This feature transforms urbs from a linear problem to a quadratic integer problem, or piecewise linear. The following graphic illustrates a process with the on/off feature and constant efficiency:



The following graphic illustrates a process with the on/off feature and part load behavior:



Coupling the throughput ant the on/off marker: The following equation introduces a coupling between $_{pt}$, the boolean on/off marker of a process and its throughput τ_{pt}, so that $_{pt}$ assumes the value 1 when the process has a non-zero output and 0 otherwise.

$$\forall p \in P^{\text{on/off}},\ t \in T_m :$$

$$\underline{P}_p \cdot \kappa_p \cdot_{pt} \leq \tau_{pt} \leq \kappa_p \cdot_{pt} + \underline{P}_p \cdot \kappa_p \cdot (1 -_{pt})$$

Input: The following equation describes the alteration of the input equation of a process with on/off and part-load behaviors due to the necessity of having a continuous, linear function defined on two intervals. The first interval represents the starting input of a process, while the second one represents the consumed

input while also producing.

$$\forall p \in P^{\text{on/off with part load}},\ c \in C,\ t \in T_m:$$

$$\epsilon^{in}_{pct} = \tau_{pt} \cdot r^{\text{in}}_{pc} \cdot (1 - {}_{pt}) + \Delta t \cdot \left(\frac{\underline{r}^{\text{in}}_{pc} - r^{\text{in}}_{pc}}{1 - \underline{P}_p} \cdot \underline{P}_p \cdot \kappa_p + \frac{r^{\text{in}}_{pc} - \underline{P}_p \underline{r}^{\text{in}}_{pc}}{1 - \underline{P}_p} \cdot \tau_{pt} \right) \cdot {}_{pt}\ .$$

In order to ensure the continuity property of the function, the input ratio used for the starting interval has to be one corresponding to the minimum partial load, using $\underline{r}^{\text{in}}_{pc}$. This is a realistic value, since processes normally use, percentagewise, more fuel in relationship to the throughput when starting than at higher throughput values.

Output differentiation: The following equations differentiate whether an output commodity needs to be produced when a process is starting (e.g. environmental commodities) or not (e.g. electricity):

$$\forall p \in P^{\text{on/off}},\ c \in C^{\text{environmental}},\ t \in T_m:$$

$$\epsilon^{out}_{pct} = \tau_{pt} \cdot r^{\text{out}}_{pc}$$

$$\forall p \in P^{\text{on/off}},\ c \in C^{\text{non-environmental}},\ t \in T_m:$$

$$\epsilon^{out}_{pct} = \tau_p t \cdot r^{\text{out}}_{pc} \cdot {}_{pt}\ .$$

If the process also shows part-load behavior, the previous two equations change to a similarly adapted version of the part-load output equation:

$$\forall p \in P^{\text{on/off with part load}},\ c \in C^{\text{environmental}},\ t \in T_m:$$

$$\epsilon^{out}_{pct} = \tau_p t \cdot r^{\text{out}}_{pc} \cdot (1 - {}_{pt}) + \Delta t \cdot \left(\frac{\underline{r}^{\text{out}}_{pc} - r^{\text{out}}_{pc}}{1 - \underline{P}_p} \cdot \underline{P}_p \cdot \kappa_p + \frac{r^{\text{out}}_{pc} - \underline{P}_p \underline{r}^{\text{out}}_{pc}}{1 - \underline{P}_p} \cdot \tau_{pt} \right) \cdot {}_{pt}$$

$$\forall p \in P^{\text{on/off}},\ c \in C^{\text{non-environmental}},\ t \in T_m:$$

$$\epsilon^{\text{out}}_{pct} = \Delta t \cdot \left(\frac{\underline{r}^{\text{out}}_{pc} - r^{\text{out}}_{pc}}{1 - \underline{P}_p} \cdot \underline{P}_p \cdot \kappa_p + \frac{r^{\text{out}}_{pc} - \underline{P}_p \underline{r}^{\text{out}}_{pc}}{1 - \underline{P}_p} \cdot \tau_{pt} \right) \cdot {}_{pt}\ .$$

Here, it is important to notice that the output of the environmental commodities becomes a continuous, piecewise linear function defined on two intervals. In order to ensure the continuity property of the function, the output ratio used for the starting interval has to be the partial one, $\underline{r}^{\text{in}}_{pc}$. This is a realistic value, since processes normaly produce, percentagewise, more CO2 and/or other environmental commodities in relationship to the throughput when starting then at higher throughput values.

Output ramping-up limit: While ramping up a process which can be turned on and off with a defined ramping up gradient, the following unrealistic situation might occur: Due to the fact that in the minimum working point the process on/off marker ${}_{pt}$ can be both 0 and 1, the output of a process might have unrealistic jumps after the starting process is completed. There are 3 possible cases, each solved with its own output ramping equation, as follows:

Case I: When

$$\underline{P}_p \geq \overline{PG}^{\text{up}}_p$$

$\underline{P}_p$ is a multiple of $\overline{PG}^{\text{up}}_p$.

Here, in order to ensure that the process behaves realistically, it is needed to ensure that the process starts producing in the minimum working point, $\underline{P}_p \kappa_p \; r^{\text{out}}_{pc}$, and not at a higher value. This is done by the following equation:

$$\forall p \in P^{\text{on/off, case I}},\; c \in C,\; t \in T_m :$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t \underline{P}_p \kappa_p r^{\text{out}}_{pc}.$$

If the process shows a part load behavior, the equation changes to:

$$\forall p \in P^{\text{on/off with part load, case I}},\; c \in C,\; t \in T_m :$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t \underline{P}_p \kappa_p \underline{r}^{\text{out}}_{pc}.$$

If the process has a time variable efficiency, the equation changes to:

$$\forall p \in P^{\text{on/off with TimeVarEff, case I}},\; c \in C,\; t \in T_m :$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t \underline{P}_p \kappa_p r^{\text{out}}_{pc} f^{\text{out}}_{pt}.$$

If the process has both a part load behavior and a time variable efficiency, the equation changes to:

$$\forall p \in P^{\text{on/off with TimeVarEff, case I}},\; c \in C,\; t \in T_m :$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t \underline{P}_p \kappa_p \underline{r}^{\text{out}}_{pc} f^{\text{out}}_{pt}.$$

Case II: When

$$\underline{P}_p > \overline{PG}^{\text{up}}_p$$
$$\underline{P}_p \text{ is not a multiple of} \overline{PG}^{\text{up}}_p.$$

Here, in order to ensure that the process behaves realistically, it is needed to ensure that the process starts somewhere in the interval between the minimum working point $\underline{P}_p \kappa_p$ and the point of the first multiple of $\overline{PG}^{\text{up}}_p$ greater than $\underline{P}_p \kappa_p$, which is $(\frac{\underline{P}_p}{\overline{PG}^{\text{up}}_p} + 1) \cdot \overline{PG}_p$, where is the rounded down number. This is done by the following equation:

$$\forall p \in P^{\text{on/off, case II}},\; c \in C,\; t \in T_m :$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t (\frac{\underline{P}_p}{\overline{PG}^{\text{up}}_p} + 1) \overline{PG}_p \kappa_p r^{\text{out}}_{pc}.$$

If the process shows a part load behavior, the equation changes to:

$$\forall p \in P^{\text{on/off, case II}},\; c \in C,\; t \in T_m :$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t (\frac{\underline{P}_p}{\overline{PG}^{\text{up}}_p} + 1) \overline{PG}_p \kappa_p \underline{r}^{\text{out}}_{pc}.$$

If the process has a time variable efficiency, the equation changes to:

$$\forall p \in P^{\text{on/off with TimeVarEff, case II}},\; c \in C,\; t \in T_m :$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t (\frac{\underline{P}_p}{\overline{PG}^{\text{up}}_p} + 1) \overline{PG}_p \kappa_p r^{\text{out}}_{pc} f^{\text{out}}_{pt}.$$

If the process has both a part load behavior and a time variable efficiency, the equation changes to:

$$\forall p \in P^{\text{on/off with part load and TimeVarEff, case II}},\ c \in C,\ t \in T_m:$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t(\frac{\underline{P}_p}{\overline{PG}^{\text{up}}_p} + 1)\overline{PG}_p \kappa_p \underline{r}^{\text{out}}_{pc} f^{\text{out}}_{pt}.$$

Case III: When

$$\underline{P}_p < \overline{PG}^{\text{up}}_p.$$

Here, in order to ensure that the process behaves realistically, it is needed to ensure that the process starts somewhere in the interval between the minimum working point $\underline{P}_p \kappa_p$ and the first ramping up point greater than 0, $\overline{PG}^{\text{up}}_p \kappa_p$. This is done by the following equation:

$$\forall p \in P^{\text{on/off, case III}},\ c \in C,\ t \in T_m:$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t \overline{PG}^{\text{up}}_p \kappa_p r^{\text{out}}_{pc}.$$

If the process shows a part load behavior, the equation changes to:

$$\forall p \in P^{\text{on/off, case III}},\ c \in C,\ t \in T_m:$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t \overline{PG}^{\text{up}}_p \kappa_p \underline{r}^{\text{out}}_{pc}.$$

If the process has a time variable efficiency, the equation changes to:

$$\forall p \in P^{\text{on/off with TimeVarEff, case III}},\ c \in C,\ t \in T_m:$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t \overline{PG}^{\text{up}}_p \kappa_p r^{\text{out}}_{pc}\ f^{\text{out}}_{pt}.$$

If the process has both a part load behavior and a time variable efficiency, the equation changes to:

$$\forall p \in P^{\text{on/off with part load and TimeVarEff, case III}},\ c \in C,\ t \in T_m:$$

$$\epsilon^{out}_{pct} - \epsilon^{out}_{pc(t-1)} \leq \Delta t \overline{PG}^{\text{up}}_p \kappa_p \underline{r}^{\text{out}}_{pc} f^{\text{out}}_{pt}.$$

Starting ramp-up: There are some processes which have a different ramping up gradient while starting than while producing. This is usually defined with the help of a so called starting time. The following equations transform the starting time into a starting ramp and implement the starting ramp only during start, either as the only ramping constraint when no ramp up gradient is defined or by replacing during start the rampiong up constraint which uses the ramping up gradient:

$$\forall p \in P^{\text{on/off with start time}},\ t \in T_m:$$

$$SR_p = \frac{\underline{P}_p}{ST_p}$$
$$\tau_{pt} - \tau_{p(t-1)} \leq \Delta t \overline{PG}^{\text{up}}_p \kappa_{pp(t-1)} + \Delta t\ SR_p \kappa_p (1 - _{p(t-1)}).$$

Start-up costs: For those processes which have a fix start-up cost, it is necessary to identify whether a process has completed its starting phase and begins to produce or not. The following equation does

this by turning the boolean variable process start-up marker σ_{pt} to 1 when the process on/off marker switches from 0 to 1:

$$\forall p \in P^{\text{on/off with start cost}},\ t \in T_m:$$

$$\sigma_{pt} \geq_{pt} -_{p(t-1)}.$$

The following table shows the possible values of σ_{pt}: .. table:: *Table: Process Start-up Marker Values*

pt	$p(t-1)$	σ_{pt}
0	0	0 or 1 (0 is optimal)
0	1	0
1	0	1
1	1	0

Costs

The cost function is ammended with one cost type, the start-up cost:

$$\zeta = \zeta_{\text{inv}} + \zeta_{\text{fix}} + \zeta_{\text{var}} + \zeta_{\text{fuel}} + \zeta_{\text{startup}} + \zeta_{\text{env}}.$$

Turning on a process requires sometime an additional fix cost besides the fuel used for the starting. As the variable costs, these costs occur when processes are used:

$$\zeta_{\text{startup}} = w\Delta t \sum_{t \in T_m}$$
$$p \in P_{\text{on/off}} P_p^{\text{start}} \sigma_{pt},$$

where P_p^{start} is the fix start-up cost and σ_{pt} is the process start-up marker. This cost type can also be merged into the same class of costs as the variable and fuel costs.

1.3 Technical documentation

Continue here if you want to understand in detail the model generator implementation.

1.3.1 Model Implementation

In this Section the **implementation** of the theoretical concepts of the model is described. This includes listing and describing all relevant sets, parameters, variables and constraints linking mathematical notation with the corresponding code fragment.

Sets

Since urbs is a linear optimization model with many objects (e.g variables, parameters), it is reasonable to use sets to define the groups of objects. With the usage of sets, many facilities are provided, such as understanding the main concepts of the model. Many objects are represented by various sets, therefore sets can be easily used to check whether some object has a specific characteristic or not. Additionally sets are useful to define a hierarchy of objects. Mathematical notation of sets are expressed with uppercase letters, and their members are usually expressed with the same lowercase letters. Main sets, tuple sets and subsets will be introduced in this respective order.

Elementary sets

Table 3: *Table: Model Sets*

Set	Description
$t \in T$	Timesteps
$t \in T_{\mathrm{m}}$	Modelled Timesteps
$y \in Y$	Support timeframes
$v \in V$	Sites
$c \in C$	Commodities
$q \in Q$	Commodity Types
$p \in P$	Processes
$s \in S$	Storages
$f \in F$	Transmissions
$r \in R$	Cost Types

Time Steps

The model urbs is considered to observe a energy system model and calculate the optimal solution within a limited span of time. This limited span of time is viewed as a discrete variable, which means values of variables are viewed as occurring only at distinct timesteps. The set of **time steps** $T = \{t_0, \ldots, t_N\}$ for N in $\mathbb{N}$ represents Time. This set contains $N+1$ sequential time steps with equal spaces. Each time step represents another point in time. At the initialisation of the model this set is fixed by the user by setting the variable `timesteps` in script `runme.py`. Duration of space between timesteps $\Delta t = t_{x+1} - t_x$, length of simulation $\Delta t \cdot N$ and time interval $[t_0, t_N]$ can be fixed to satisfy the needs of the user. In code this set is defined by the set `t` and initialized by the section:

```
m.t = pyomo.Set(
    initialize=m.timesteps,
    ordered=True,
    doc='Set of timesteps')
```

Where:

- *Initialize*: A function that receives the set indices and model to return the value of that set element, initializes the set with data.
- *Ordered*: A boolean value that indicates whether the set is ordered.
- *Doc*: A string describing the set.

Modelled Timesteps

The Set, **modelled timesteps**, is a subset of the time steps set. The only difference between modelled timesteps set and the timesteps set is that the initial timestep t_0 is not included. All other features of the set time steps also apply to the set of modelled timesteps. This set is the main time set used in the model. The distinction with the set **timesteps** is only required to facilitate the definition of the storage state equation. In script `model.py` this set is defined by the set `tm` and initialized by the code fragment:

```
m.tm = pyomo.Set(
    within=m.t,
    initialize=m.timesteps[1:],
    ordered=True,
    doc='Set of modelled timesteps')
```

Where:

- *Within*: The option that supports the validation of a set array.
- `m.timesteps[1:]` represents the timesteps set starting from the second element, excluding the first timestep t_0

Support timeframes

Support timeframes are represented by the set Y. They represent the explicitly modeled support timeframes, e.g., years, for intertemporal models. In script `model.py` the set is defined as:

```
m.stf = pyomo.Set(
    initialize=(m.commodity.index.get_level_values('support_timeframe')
                .unique()),
    doc='Set of modeled support timeframes (e.g. years)')
```

Sites

Sites are represented by the set V. A Site v can be any distinct location, a place of settlement or activity (e.g *process*, *transmission*, *storage*).A site is for example an individual building, region, country or even continent. Sites can be imagined as nodes(vertices) on a graph of locations, connected by edges. Index of this set are the descriptions of the Sites (e.g north, middle, south). In script `model.py` this set is defined by `sit` and initialized by the code fragment:

```
m.sit = pyomo.Set(
    initialize=m.commodity.index.get_level_values('Site').unique(),
    doc='Set of sites')
```

Commodities

As explained in the Overview section, **commodities** are goods that can be generated, stored, transmitted or consumed. The set of Commodities represents all goods that are relevant to the modelled energy system, such as all energy carriers, inputs, outputs, intermediate substances. (e.g Coal, CO2, Electric, Wind) By default, commodities are given by their energy content (MWh). Usage of some commodities are limited by a maximum value or maximum value per timestep due to their availability or restrictions, also some commodities have a price that needs to be compensated..(e.g coal, wind, solar).In script `model.py` this set is defined by `com` and initialized by the code fragment:

```
m.com = pyomo.Set(
    initialize=m.commodity.index.get_level_values('Commodity').unique(),
    doc='Set of commodities')
```

Commodity Types

Commodities differ in their usage purposes, consequently **commodity types** are introduced to subdivide commodities by their features. These Types are hard coded as `SupIm`, `Stock`, `Demand`, `Env`, `Buy`, `Sell`. In script `model.py` this set is defined as `com_type` and initialized by the code fragment:

```
m.com_type = pyomo.Set(
    initialize=m.commodity.index.get_level_values('Type').unique(),
    doc='Set of commodity types')
```

Processes

One of the most important elements of an energy system is the **process**. A process p can be defined by the action of changing one or more forms of energy, i.e. commodities, to others. In our modelled energy system, processes convert input commodities into output commodities. Process technologies are represented by the set processes P. Different processes technologies have fixed input and output commodities. These input and output commodities can be either single or multiple regardless of each other. Some example members of this set can be: *Wind Turbine*,'Gas Plant', *Photovoltaics*. In script `model.py` this set is defined as `pro` and initialized by the code fragment:

```
m.pro = pyomo.Set(
    initialize=m.process.index.get_level_values('Process').unique(),
    doc='Set of conversion processes')
```

Storages

Energy **Storage** is provided by technical facilities that store energy to generate a commodity at a later time for the purpose of meeting the demand. Occasionally, on-hand commodities may not be able to satisfy the required amount of energy to meet the demand, or the available amount of energy may be much more than required.Storage technologies play a major role in such circumstances. The Set S represents all storage technologies (e.g *Pump storage*). In script `model.py` this set is defined as `sto` and initialized by the code fragment:

```
m.sto = pyomo.Set(
    initialize=m.storage.index.get_level_values('Storage').unique(),
    doc='Set of storage technologies')
```

Transmissions

Transmissions $f \in F$ represent possible conveyances of commodities between sites. Transmission process technologies can vary between different commodities, due to distinct physical attributes and forms of commodities. Some examples for Transmission technologies are: *hvac*, *hvdc*, *pipeline*) In script `model.py` this set is defined as `tra` and initialized by the code fragment:

```
m.tra = pyomo.Set(
    initialize=m.transmission.index.get_level_values('Transmission').
↪unique(),
    doc='Set of transmission technologies')
```

Cost Types

One of the major goals of the model is to calculate the costs of a simulated energy system. There are 6 different types of costs. Each one has different features and are defined for different instances. Set of **cost types** is hardcoded, which means they are not considered to be fixed or changed by the user. The Set R defines the Cost Types, each member r of this set R represents a unique cost type name. The cost types are hard coded as: `Investment`, `Fix`, `Variable`, `Fuel`, `Revenue`, `Purchase`, `Startup`. In script `model.py` this set is defined as `cost_type` and initialized by the code fragment:

```
m.cost_type = pyomo.Set(
    initialize=['Inv', 'Fix', 'Var', 'Fuel','Revenue','Purchase','Startup
↪'],
    doc='Set of cost types (hard-coded)')
```

Tuple Sets

A tuple is finite, ordered collection of elements. For example, the tuple `(hat,red,large)` consists of 3 ordered elements and defines another element itself. Tuples are needed in this model to define the combinations of elements from different sets. Defining a tuple lets us assemble related elements and use them as a single element. These tuples are then collected into tuple sets. These tuple sets are then immensely useful for efficient indexing of model variables and parameters and for defining the constraint rules.

Commodity Tuples

Commodity tuples represent combinations of defined commodities. These are represented by the set C_{yvq}. A member c_{yvq} in set C_{yvq} is a commodity c of commodity type q in support timeframe y and site v. For example, *(2020, Mid, Elec, Demand)* is interpreted as commodity *Elec* of commodity type *Demand* in the year *2020* in site *Mid*. This set is defined as `com_tuples` and given by the code fragment:

```
m.com_tuples = pyomo.Set(
    within=m.stf*m.sit*m.com*m.com_type,
    initialize=m.commodity.index,
    doc='Combinations of defined commodities, e.g. (2020,Mid,Elec,Demand)')
```

Process Tuples

Process tuples represent possible placements of processes within the model. These are represented by the set P_v. A member p_{yv} in set P_{yv} is a process p in support timeframe y and site v. For example, *(2020, North, Coal Plant)* is interpreted as process *Coal Plant* in site *North* in the year *2020*. This set is defined as `pro_tuples` and given by the code fragment:

```
m.pro_tuples = pyomo.Set(
    within=m.stf*m.sit*m.pro,
    initialize=m.process.index,
    doc='Combinations of possible processes, e.g. (2020,North,Coal plant)')
```

There are several subsets defined for process tuples, which each activate a different set of modeling constraints.

The first subset is formed in order to capture all processes that take up a certain area and are thus subject to the area constraint at the given site. These processes are identified by the parameter `area-per-cap` set in table *Process*, if at the same time a value for `area` is set in table *Site*. The tuple set is defined as:

```
m.pro_area_tuples = pyomo.Set(
    within=m.stf*m.sit*m.pro,
    initialize=m.proc_area.index,
    doc='Processes and Sites with area Restriction')
```

The second subset is formed in order to capture all processes which have the parameter process new capacity block `cap-block` K_{yvp}^{block} set in the table *Process*, used for building new capacity in blocks. The tuple set is defined as:

```
m.pro_cap_new_block_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro,
    initialize=[(stf, site, process)
                for (stf, site, process) in m.pro_tuples
                for (s, si, pro) in tuple(m.cap_block_dict.keys())
                if process == pro and si == site and s == stf],
                doc='Processes with new capacities built in blocks')
```

The third subset of the process tuples `pro_minfraction_tuples` $P_{yv}^{\text{minfraction}}$ is formed in order to identify processes that have a minimum fraction defined without having partial operation properties and cannot be turned off. Programatically, they are identified by those processes which have the parameter `min-fraction` set and the parameter `on-off` set to 0 in the table *Process*. The tuple set is defined in AdvancedProcesses.py as:

```
m.pro_minfraction_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro,
    initialize=[(stf, site, process)
                for (stf, site, process) in m.pro_tuples
                for (st, sit, pro) in tuple(m.min_fraction_dict.keys())
                if  stf == st and sit == site and process ==pro and
                m.process_dict['on-off'][stf, site, process] != 1],
    doc='Processes with constant efficiency and minimum working load which'
        'cannot be turned off')
```

The fourth subset of the process tuples `pro_partial_tuples` P_{yv}^{partial} is formed in order to identify processes that have partial operation properties and cannot be turned off. Programmatically, they are identified by those processes, which have the parameter `ratio-min` set for one of their input and/or outputcommodities in table *Process-Commodity* and the parameter `on-off` in the table *Process* set to 0. The tuple set is defined in AdvancedProcesses.py as:

```
m.pro_partial_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro,
    initialize=[(stf, site, process)
                for (stf, site, process) in m.pro_tuples
                for (s, pro, _) in tuple(m.r_in_min_fraction_dict.keys() or
                                         m.r_out_min_fraction_dict.keys())
                if process == pro and s == stf and
                m.process_dict['on-off'][stf, site, process] != 1],
    doc='Processes with partial input/output which cannot be turned off')
```

The fifth subset of the process tuples `pro_on_off_tuples` $P_{yv}^{\text{on/off}}$ is formed in order to identify processes that have a minimum fraction defined without having partial operation properties and can be turned off. Programatically, they are identified by those processes which have the parameter `min-fraction` set and the parameter `on-off` set to 1 in the table *Process*. The tuple set is defined in AdvancedProcesses.py as:

```
m.pro_on_off_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro,
    initialize=[(stf, site, process)
                for (stf, site, process) in
                                        tuple(m.min_fraction_dict.keys())
                for (st, sit, pro) in tuple(m.onoff_dict.keys())
                if stf == st and site == sit and process == pro],
    doc='Processes with minimal fraction which can be turned off')
```

The sixth subset of the process tuples `pro_on_off_partial_tuples` $P_{yv}^{\text{partial on/off}}$ is formed in order to identify processes that have a minimum fraction defined, partial operation properties and can be turned off. Programmatically, they are identified by those processes, which have the parameter `ratio-min` set for one of their input and/or outputcommodities in table *Process-Commodity* and the parameter `on-off` in the table *Process* set to 1. The tuple set is defined in AdvancedProcesses.py as:

```
m.pro_partial_on_off_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro,
    initialize=[(stf, site, process)
                for (stf, site, process) in m.pro_tuples
                for (st, pro, _) in tuple(m.r_in_min_fraction_dict.keys()
                                          or m.r_out_min_fraction_dict)
                if process == pro and stf == st and
                m.process_dict['on-off'][stf, site, process] == 1],
    doc='Processes with partial input/output which can be turned off')
```

Finally, processes that are subject to restrictions in the change of operational state are captured with the `pro_rampupgrad_tuples` and `pro_rampdowngrad_tuples`. This subsets are defined in AdvancedProcesses as:

```
m.pro_rampupgrad_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro,
    initialize=[(stf, sit, pro)
                for (stf, sit, pro) in m.pro_tuples
                if m.process_dict['ramp-up-grad'][stf, sit, pro] < 1.0 /␣
↪dt],
    doc='Processes with maximum ramp up gradient smaller than timestep␣
↪length')
```

```
m.pro_rampdowngrad_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro,
    initialize=[(stf, sit, pro)
                for (stf, sit, pro) in m.pro_tuples
                if m.process_dict['ramp-down-grad'][stf, sit, pro] < 1.0 /␣
↪dt],
    doc='Processes with maximum ramp down gradient smaller than timestep␣
↪length')
```

In the case of a a process which can be turned on and off and are subject to restrictions in the change of operational state while starting are captured with the `pro_rampup_start_tuples`, subset which is defined in advancedProcesses.py as:

```
m.pro_rampup_start_tuples = pyomo.Set(
        within=m.stf * m.sit *m.pro,
        initialize=[(stf, sit, pro)
                    for (stf, sit, pro) in m.pro_on_off_tuples
                    if m.process_dict['start-time'][stf, sit, pro]
                                                        > 1.0 / m.dt],
        doc='Processes with different starting ramp up gradient')
```

Transmission Tuples

Transmission tuples represent possible transmissions. These are represented by the set $F_{ycv_{\text{out}}v_{\text{in}}}$. A member $f_{ycv_{\text{out}}v_{\text{in}}}$ in set $F_{ycv_{\text{out}}v_{\text{in}}}$ is a transmission f,that is directed from an origin site v_{out} to a destination site v_{in} and carrying the commodity c in support timeframe y. The term "*directed from an origin site* v_{out} *to a destination site* v_{in}" can also be defined as an arc a . For example, *(2020, South, Mid, hvac, Elec)* is interpreted as transmission *hvac* that is directed from origin site *South* to destination site *Mid* carrying commodity *Elec* in year *2020*. This set is defined as `tra_tuples` and given by the code fragment:

```
m.tra_tuples = pyomo.Set(
    within=m.stf*m.sit*m.sit*m.tra*m.com,
    initialize=m.transmission.index,
    doc='Combinations of possible transmissions, e.g. '
        '(2020,South,Mid,hvac,Elec)')
```

The set $F^{blocks}_{ycv_{\text{out}}v_{\text{in}}}$ includes all transmission lines which have a defined capacity block for the building of new transmission capacities.

```
m.tra_block_tuples = pyomo.Set(
    within=m.stf * m.sit * m.sit * m.tra * m.com,
    initialize=[(stf, sit, sit_, tra, com)
                for (stf, sit, sit_, tra, com) in tuple(m.tra_block_dict.
↪keys())],
    doc='Transmission with new block capacities')
```

DCPF Transmission Tuples

If the DC Power Flow Model feature is activated in the model, three different transmission tuple sets are defined in the model.

The set $F_{ycv_{\text{out}}v_{\text{in}}TP}$ includes every transport model transmission lines and is defined as `tra_tuples_tp` and given by the code fragment:

```
m.tra_tuples_tp = pyomo.Set(
    within=m.stf * m.sit * m.sit * m.tra * m.com,
    initialize=tuple(tra_tuples_tp),
    doc='Combinations of possible transport transmissions,'
        'e.g. (2020,South,Mid,hvac,Elec)')
```

The set $F_{ycv_{\text{out}}v_{\text{in}}DCPF}$ includes every transmission line, which should be modelled with DCPF. If the complementary arcs are included in the input for DCPF transmission lines, these will be excluded from this set with `remove_duplicate_transmission()`. This set is defined as `tra_tuples_dc` and given by the code fragment:

```
m.tra_tuples_dc = pyomo.Set(
    within=m.stf * m.sit * m.sit * m.tra * m.com,
    initialize=tuple(tra_tuples_dc),
    doc='Combinations of possible bidirectional dc'
        'transmissions, e.g. (2020,South,Mid,hvac,Elec)')
```

If the DCPF is activated, the set $F_{ycv_{\text{out}}v_{\text{in}}}$ is defined by the unification of the sets $F_{ycv_{\text{out}}v_{\text{in}}DCPF}$ and $F_{ycv_{\text{out}}v_{\text{in}}TP}$. This set is defined as `tra_tuples` in the same fashion as the default transmission model.

Storage Tuples

Storage tuples label storages. They are represented by the set S_{yvc}. A member s_{yvc} in set S_{yvc} is a storage s of commodity c in site v and support timeframe y For example, *(2020, Mid, Bat, Elec)* is interpreted as storage *Bat* for commodity *Elec* in site *Mid* in the year *2020*. This set is defined as `sto_tuples` and given by the code fragment:

```
m.sto_tuples = pyomo.Set(
    within=m.stf*m.sit*m.sto*m.com,
    initialize=m.storage.index,
    doc='Combinations of possible storage by site,'
        'e.g. (2020,Mid,Bat,Elec)')
```

There are four subsets of storage tuples.

In a first subset of the storage tuples are all storages that have a user defined fixed value for the initial state are collected.

```
m.sto_init_bound_tuples = pyomo.Set(
    within=m.stf*m.sit*m.sto*m.com,
    initialize=m.stor_init_bound.index,
    doc='storages with fixed initial state')
```

A second subset is defined for all storages that have a fixed ratio between charging/discharging power and storage content.

```
m.sto_ep_ratio_tuples = pyomo.Set(
    within=m.stf*m.sit*m.sto*m.com,
    initialize=tuple(m.sto_ep_ratio_dict.keys()),
    doc='storages with given energy to power ratio')
```

The third and fourth subsets are defined for all the storages that have a capacity or power expansion block defined in the input.

```
m.sto_block_c_tuples = pyomo.Set(
    within=m.stf * m.sit * m.sto * m.com,
    initialize=tuple(m.sto_block_c_dict.keys()),
    doc='storages with new energy block capacities')
m.sto_block_p_tuples = pyomo.Set(
    within=m.stf * m.sit * m.sto * m.com,
    initialize=tuple(m.sto_block_p_dict.keys()),
    doc='storages with new power block capacities')
```

Process Input Tuples

Process input tuples represent commodities consumed by processes. These are represented by the set C^{in}_{yvp}. A member c^{in}_{yvp} in set C^{in}_{vp} is a commodity c consumed by the process p in site v in support timeframe y. For example, *(2020, Mid, PV, Solar)* is interpreted as commodity *Solar* consumed by the process *PV* in the site *Mid* in the year *2020*. This set is defined as `pro_input_tuples` and given by the code fragment:

```
m.pro_input_tuples = pyomo.Set(
    within=m.stf*m.sit*m.pro*m.com,
    initialize=[(stf, site, process, commodity)
                for (stf, site, process) in m.pro_tuples
                for (s, pro, commodity) in m.r_in.index
                if process == pro and s == stf],
    doc='Commodities consumed by process by site,'
        'e.g. (2020,Mid,PV,Solar)')
```

Where: `r_in` represents the process input ratio as set in the input.

For processes in the tuple set `pro_partial_tuples`, the following tuple set `pro_partial_input_tuples` enumerates their input commodities. It is used to index the constraints that modifies a process' input commodity flow with respect to the standard case without partial operation. It is defined by the following code fragment:

```
m.pro_partial_input_tuples = pyomo.Set(
    within=m.stf*m.sit*m.pro*m.com,
    initialize=[(stf, site, process, commodity)
                for (stf, site, process) in m.pro_partial_tuples
                for (s, pro, commodity) in m.r_in_min_fraction.index
                if process == pro and s == stf],
    doc='Commodities with partial input ratio,'
        'e.g. (2020,Mid,Coal PP,Coal)')
```

Where: `r_in_min_fraction` represents the process input ratio as set in the input for the minimum load of the process.

For processes in the tuple set `pro_on_off_tuples`, the following tuple set `pro_on_off_input_tuples` enumerates their input commodities. It is used to index the constraints that modifies a process' input commodity flow with respect to the standard case without the on/off feature. It is defined by the following code fragment in AdvancedProcesses.py:

```
m.pro_on_off_input_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro * m.com,
    initialize=[(stf, site, process, commodity)
                for (stf, site, process) in m.pro_on_off_tuples
                for (s, pro, commodity) in tuple(m.r_in_dict.keys())
                if process == pro and stf == s],
    doc='Commodities for on/off input')
```

For processes in the tuple set `pro_partial_on_off_tuples`, the following tuple set `pro_partial_on_off_input_tuples` enumerates their input commodities. It is used to index the constraints that modifies a process' input commodity flow with respect to the standard case without the on/off feature and partial operation. It is defined by the following code fragment in AdvancedProcesses.py:

```
m.pro_partial_on_off_input_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro * m.com,
    initialize=[(stf, site, process, commodity)
                for (stf, site, process) in m.pro_partial_on_off_tuples
                for (s, pro, commodity) in tuple(m.r_in_min_fraction_dict
                                                 .keys())
                if process == pro and s == stf],
    doc='Commodities with partial input ratio which can be turned off,'
            'e.g. (2020,Mid,Coal PP,Coal)')
```

Process Output Tuples

Process output tuples represent commodities generated by processes. These are represented by the set C^{out}_{yvp}. A member c^{out}_{yvp} in set C^{out}_{yvp} is a commodity c generated by the process p in site v and support timeframe y. For example, *(2020, Mid,PV,Elec)* is interpreted as the commodity *Elec* is generated by the process *PV* in the site *Mid* in the year *2020*. This set is defined as `pro_output_tuples` and given by the code fragment:

```
m.pro_output_tuples = pyomo.Set(
    within=m.stf*m.sit*m.pro*m.com,
    initialize=[(stf, site, process, commodity)
                for (stf, site, process) in m.pro_tuples
                for (s, pro, commodity) in m.r_out.index
                if process == pro and s == stf],
    doc='Commodities produced by process by site, e.g. (2020,Mid,PV,Elec)')
```

Where: `r_out` represents the process output ratio as set in the input.

There are several alternative tuple sets that are active whenever their respective features are set in the input.

First, for processes in the tuple set `pro_partial_tuples`, the tuple set `pro_partial_output_tuples` enumerates their output commodities. It is used to index the constraints that modifies a process' output commodity flow with respect to the standard case without partial operation. It is defined by the following code fragment:

```
m.pro_partial_output_tuples = pyomo.Set(
    within=m.stf*m.sit*m.pro*m.com,
    initialize=[(stf, site, process, commodity)
                for (stf, site, process) in m.pro_partial_tuples
                for (s, pro, commodity) in m.r_out_min_fraction.index
                if process == pro and s == stf],
    doc='Commodities with partial input ratio, e.g. (Mid,Coal PP,CO2)')
```

Second, for processes in the tuple set `pro_on_off_tuples`, the tuple set `pro_on_off_output_tuples` enumerates their output commodities. It is used to index the constraints that modifies a process' output commodity flow with respect to the standard case without the on/off feature. It is defined by the following code fragment in AdvancedProcesses.py:

```
m.pro_on_off_output_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro * m.com,
    initialize=[(stf, site, process, commodity)
                for (stf, site, process) in m.pro_on_off_tuples
```

(continues on next page)

(continued from previous page)

```
                for (s, pro, commodity) in tuple(m.r_out_dict.keys())
                if process == pro and stf == s],
    doc='Commodities for on/off output')
```

Third, for processes in the tuple set `pro_partial_on_off_tuples`, the tuple set `pro_partial_on_off_output_tuples` enumerates their output commodities. It is used to index the constraints that modifies a process' output commodity flow with respect to the standard case without the on/off feature and partial operation. It is defined by the following code fragment in AdvancedProcesses.py:

```
m.pro_partial_on_off_output_tuples = pyomo.Set(
        within=m.stf * m.sit * m.pro * m.com,
        initialize=[(stf, site, process, commodity)
                    for (stf, site, process) in m.pro_partial_on_off_tuples
                    for (s, pro, commodity) in tuple(m.r_out_min_fraction_
↪dict
                                                     .keys())
                    if process == pro and s == stf],
        doc='Commodities for on/off output with partial behaviour')
```

Fourth, the processes in the tuple sets `pro_on_off_tuples` and `pro_partial_on_off_tuples` require another constraint to limit the excessive growth of the output of a process. This is required due to the fact that in the point of minimum load, without these limiting constraints, the process on/off marker $_{yvpt}$ can be both on and off. There are three cases to be considered:

The first case is represented by the tuple set `pro_rampup_divides_minfraction_output_tuples`, which covers the outputs of the processes for which the defined ramp up gradient and is smaller than the minimum load fraction and is a divisor of it. It is defined by the following code fragment in AdvancedProcesses.py:

```
m.pro_rampup_divides_minfraction_output_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro * m.com,
    initialize=[(stf, sit, pro, com)
                for (stf, sit, pro, com) in m.pro_on_off_output_tuples
                if m.process_dict['ramp-up-grad'][stf, sit, pro] < 1.0 / m.
↪dt and
                   m.process_dict['ramp-up-grad'][stf, sit, pro] <=
                   m.min_fraction_dict[stf, sit, pro] and
                   m.min_fraction_dict[stf, sit, pro] %
                   m.process_dict['ramp-up-grad'][stf, sit, pro] == 0 and
                   com not in m.com_env],
    doc='Output commodities of processes with ramp-up-grad smaller than'
        'timestep length and smaller equal than min-fraction and is a '
        'divisor of min-fraction')
```

The second case is represented by the tuple set `pro_rampup_not_divides_minfraction_output_tuples`, which covers the outputs of the processes for which the defined ramp up gradient and is smaller than the minimum load fraction and is not a divisor of it. It is defined by the following code fragment in AdvancedProcesses.py:

```
m.pro_rampup_not_divides_minfraction_output_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro * m.com,
    initialize=[(stf, sit, pro, com)
```

(continues on next page)

(continued from previous page)

```
                for (stf, sit, pro, com) in m.pro_on_off_output_tuples
                if m.process_dict['ramp-up-grad'][stf, sit, pro] < 1.0 / m.
↪dt and
                    m.process_dict['ramp-up-grad'][stf, sit, pro] <
                    m.min_fraction_dict[stf, sit, pro] and
                    m.min_fraction_dict[stf, sit, pro] %
                    m.process_dict['ramp-up-grad'][stf, sit, pro] != 0 and
                    com not in m.com_env],
    doc='Output commodities of processes with ramp-up-grad smaller than'
        'timestep length and smaller than min-fraction and is NOT a '
        'divisor of min-fraction')
```

The third and last case is represented by the tuple set `pro_rampup_bigger_minfraction_output_tuples`, which covers the outputs of the processes for which the defined ramp up gradient and is greater than the minimum load fraction. It is defined by the following code fragment in AdvancedProcesses.py:

```
m.pro_rampup_bigger_minfraction_output_tuples = pyomo.Set(
    within=m.stf * m.sit * m.pro * m.com,
    initialize=[(stf, sit, pro, com)
                for (stf, sit, pro, com) in m.pro_on_off_output_tuples
                if m.process_dict['ramp-up-grad'][stf, sit, pro] < 1.0 / m.
↪dt and
                    m.process_dict['ramp-up-grad'][stf, sit, pro] >
                    m.min_fraction_dict[stf, sit, pro] and
                    com not in m.com_env],
    doc='Output commodities of processes with ramp up gradient smaller'
        'than timestep length and greater than min-fraction')
```

Last, the output of all processes that have a time dependent efficiency are collected in an additional tuple set. The set contains all outputs corresponding to processes that are specified as column indices in the input file worksheet `TimeVarEff`.

```
m.pro_timevar_output_tuples = pyomo.Set(
    within=m.sit*m.pro*m.com,
    initialize=[(site, process, commodity)
                for (site, process) in m.eff_factor.columns.values
                for (pro, commodity) in m.r_out.index
                if process == pro],
    doc='Outputs of processes with time dependent efficiency')
```

Demand Side Management Tuples

There are two kinds of demand side management (DSM) tuples in the model: DSM site tuples D_{yvc} and DSM down tuples $D^{\text{down}}_{yvct,tt}$. The first kind D_{yvc} represents all possible combinations of support timeframe y, site v and commodity c of the DSM sheet. It is given by the code fragment:

```
m.dsm_site_tuples = pyomo.Set(
    within=m.stf*m.sit*m.com,
    initialize=m.dsm.index,
    doc='Combinations of possible dsm by site, e.g. (2020, Mid, Elec)')
```

The second kind $D^{\text{down}}_{t,tt,yvc}$ refers to all possible DSM downshift possibilities. It is defined to overcome the difficulty caused by the two time indices of the DSM downshift variable. Dependend on support

timeframe y, site v and commodity c the tuples contain two time indices. For example *(5001, 5003, 2020, Mid, Elec)* is intepreted as the downshift in timestep *5003*, which was caused by the upshift of timestep *5001* in year *2020 and 'site 'Mid* for commodity *Elec*. The tuples are given by the following code fragment:

```
m.dsm_down_tuples = pyomo.Set(
    within=m.tm*m.tm*m.stf*m.sit*m.com,
    initialize=[(t, tt, stf, site, commodity)
                for (t, tt, stf, site, commodity)
                in dsm_down_time_tuples(m.timesteps[1:],
                                        m.dsm_site_tuples,
                                        m)],
    doc='Combinations of possible dsm_down combinations, e.g. '
        '(5001,5003,2020,Mid,Elec)')
```

where the following function is utilized:

```
def dsm_down_time_tuples(time, sit_com_tuple, m):
    """ Dictionary for the two time instances of DSM_down
    Args:
        time: list with time indices
        sit_com_tuple: a list of (site, commodity) tuples
        m: model instance
    Returns:
        A list of possible time tuples depending on site and commodity
    """

    delay = m.dsm_dict['delay']
    ub = max(time)
    lb = min(time)
    time_list = []

    for (stf, site, commodity) in sit_com_tuple:
        for step1 in time:
            for step2 in range(step1 -
                               max(int(delay[stf, site, commodity] /
                                   m.dt.value), 1),
                               step1 +
                               max(int(delay[stf, site, commodity] /
                                   m.dt.value), 1) + 1):
                if lb <= step2 <= ub:
                    time_list.append((step1, step2, stf, site, commodity))

    return time_list
```

Commodity Type Subsets

Commodity Type Subsets represent the commodity tuples only from a given commodity type. Commodity Type Subsets are subsets of the sets commodity tuples These subsets can be obtained by fixing the commodity type q to a desired commodity type (e.g SupIm, Stock) in the set commodity tuples C_{vq}. Since there are 6 types of commodity types, there are also 6 commodity type subsets. Commodity type subsets are;

Supply Intermittent Commodities (`SupIm`): The set C_{sup} represents all commodities c of commodity type `SupIm`. Commodities of this type have intermittent timeseries, in other words, availability of these

commodities are not constant. These commodities might have various energy content for every timestep t. For example solar radiation is contingent on many factors such as sun position, weather and varies permanently.

Stock Commodities (`Stock`): The set C_{st} represents all commodities c of commodity type `Stock`. Commodities of this type can be purchased at any time for a given price(k_{vc}^{fuel}).

Sell Commodities (`Sell`): The set C_{sell} represents all commodities c of commodity type `Sell`. Commodities that can be sold. These Commodities have a sell price (k_{vct}^{bs}) that may vary with the given timestep t.

Buy Commodities (`Buy`): The set C_{buy} represents all commodities c of commodity type `Buy`. Commodities that can be purchased. These Commodities have a buy price (k_{vc}^{bs}) that may vary with the given timestep t.

Demand Commodities (`Demand`): The set C_{dem} represents all commodities c of commodity type `Demand`. Commodities of this type are the requested commodities of the energy system. They are usually the end product of the model (e.g Electricity:Elec).

Environmental Commodities (`Env`): The set C_{env} represents all commodities c of commodity type `Env`. Commodities of this type are usually the undesired byproducts of processes that might be harmful for environment, optional maximum creation limits can be set to control the generation of these commodities (e.g Greenhouse Gas Emissions: CO_2).

Commodity Type Subsets are given by the code fragment:

```
m.com_supim = pyomo.Set(
    within=m.com,
    initialize=commodity_subset(m.com_tuples, 'SupIm'),
    doc='Commodities that have intermittent (timeseries) input')
m.com_stock = pyomo.Set(
    within=m.com,
    initialize=commodity_subset(m.com_tuples, 'Stock'),
    doc='Commodities that can be purchased at some site(s)')
m.com_sell = pyomo.Set(
   within=m.com,
   initialize=commodity_subset(m.com_tuples, 'Sell'),
   doc='Commodities that can be sold')
m.com_buy = pyomo.Set(
    within=m.com,
    initialize=commodity_subset(m.com_tuples, 'Buy'),
    doc='Commodities that can be purchased')
m.com_demand = pyomo.Set(
    within=m.com,
    initialize=commodity_subset(m.com_tuples, 'Demand'),
    doc='Commodities that have a demand (implies timeseries)')
m.com_env = pyomo.Set(
    within=m.com,
    initialize=commodity_subset(m.com_tuples, 'Env'),
    doc='Commodities that (might) have a maximum creation limit')
```

Where:

urbs.**commodity_subset**(*com_tuples*, *type_name*)

Returns the commodity names(c) of the given commodity type(q).

Parameters

- **com_tuples** – A list of tuples (site, commodity, commodity type)

- **type_name** – A commodity type or a list of commodity types

Returns The set (unique elements/list) of commodity names of the desired commodity type.

Operational state tuples

For intertemporal optimization the operational state of units in a support timeframe *y* has to be calculated from both the initially installed units and their remaining lifetime and the units installed in a previous support timeframe which are still operational in *y*. This is achieved via 6 tuple sets two each for processes, transmissions and storages.

Intially installed units

Processes which are already installed at the beginning of the modeled time horizon and still operational in support timeframe *stf* are collected in the following tuple set:

```
m.inst_pro_tuples = pyomo.Set(
    within=m.sit*m.pro*m.stf,
    initialize=[(sit, pro, stf)
                for (sit, pro, stf)
                in inst_pro_tuples(m)],
    doc=' Installed processes that are still operational through stf')
```

where the following function is utilized:

```
def inst_pro_tuples(m):
    """ Tuples for operational status of already installed units
    (processes, transmissions, storages) for intertemporal planning.
    Only such tuples where the unit is still operational until the next
    support time frame are valid.
    """
    inst_pro = []
    sorted_stf = sorted(list(m.stf))

    for (stf, sit, pro) in m.inst_pro.index:
        for stf_later in sorted_stf:
            index_helper = sorted_stf.index(stf_later)
            if stf_later == max(m.stf):
                if (stf_later +
                   m.global_prop.loc[(max(sorted_stf), 'Weight'), 'value']␣
↪-
                   1 < min(m.stf) + m.process_dict['lifetime'][
                                                (stf, sit, pro)]):
                    inst_pro.append((sit, pro, stf_later))
            elif (sorted_stf[index_helper+1] <=
                  min(m.stf) + m.process_dict['lifetime'][(stf, sit,␣
↪pro)]):
                inst_pro.append((sit, pro, stf_later))

    return inst_pro
```

Transmissions which are already installed at the beginning of the modeled time horizon and still operational in support timeframe *stf* are collected in the following tuple set:

```
m.inst_tra_tuples = pyomo.Set(
    within=m.sit*m.sit*m.tra*m.com*m.stf,
    initialize=[(sit, sit_, tra, com, stf)
                for (sit, sit_, tra, com, stf)
                in inst_tra_tuples(m)],
    doc='Installed transmissions that are still operational through stf')
```

where the following function is utilized:

```
def inst_tra_tuples(m):
    """ s.a. inst_pro_tuples
    """
    inst_tra = []
    sorted_stf = sorted(list(m.stf))

    for (stf, sit1, sit2, tra, com) in m.inst_tra.index:
        for stf_later in sorted_stf:
            index_helper = sorted_stf.index(stf_later)
            if stf_later == max(m.stf):
                if (stf_later +
                    m.global_prop_dict['value'][(max(sorted_stf), 'Weight
↪')] -
                    1 < min(m.stf) + m.transmission_dict['lifetime'][
                        (stf, sit1, sit2, tra, com)]):
                    inst_tra.append((sit1, sit2, tra, com, stf_later))
            elif (sorted_stf[index_helper + 1] <= min(m.stf) +
                  m.transmission_dict['lifetime'][
                      (stf, sit1, sit2, tra, com)]):
                inst_tra.append((sit1, sit2, tra, com, stf_later))

    return inst_tra
```

Storages which are already installed at the beginning of the modeled time horizon and still operational in support timeframe *stf* are collected in the following tuple set:

```
m.inst_sto_tuples = pyomo.Set(
    within=m.sit*m.sto*m.com*m.stf,
    initialize=[(sit, sto, com, stf)
                for (sit, sto, com, stf)
                in inst_sto_tuples(m)],
    doc='Installed storages that are still operational through stf')
```

where the following function is utilized:

```
def inst_sto_tuples(m):
    """ s.a. inst_pro_tuples
    """
    inst_sto = []
    sorted_stf = sorted(list(m.stf))

    for (stf, sit, sto, com) in m.inst_sto.index:
        for stf_later in sorted_stf:
            index_helper = sorted_stf.index(stf_later)
            if stf_later == max(m.stf):
                if (stf_later +
                    m.global_prop_dict['value'][(max(sorted_stf), 'Weight
↪')] -
```

(continues on next page)

(continued from previous page)

```
                    1 < min(m.stf) +
                        m.storage_dict['lifetime'][(stf, sit, sto, com)]):
                    inst_sto.append((sit, sto, com, stf_later))
            elif (sorted_stf[index_helper + 1] <=
                  min(m.stf) + m.storage_dict['lifetime'][
                                (stf, sit, sto, com)]):
                inst_sto.append((sit, sto, com, stf_later))

    return inst_sto
```

Installation in earlier support timeframe

Processes installed in an earlier support timeframe *stf* and still usable in support timeframe *stf_later* are collected in the following tuple set:

```
m.operational_pro_tuples = pyomo.Set(
    within=m.sit*m.pro*m.stf*m.stf,
    initialize=[(sit, pro, stf, stf_later)
                for (sit, pro, stf, stf_later)
                in op_pro_tuples(m.pro_tuples, m)],
    doc='Processes that are still operational through stf_later'
        '(and the relevant years following), if built in stf'
        'in stf.')
```

where the following function is utilized:

```
def op_pro_tuples(pro_tuple, m):
    """ Tuples for operational status of units (processes, transmissions,
    storages) for intertemporal planning.
    Only such tuples where the unit is still operational until the next
    support time frame are valid.
    """
    op_pro = []
    sorted_stf = sorted(list(m.stf))

    for (stf, sit, pro) in pro_tuple:
        for stf_later in sorted_stf:
            index_helper = sorted_stf.index(stf_later)
            if stf_later == max(sorted_stf):
                if (stf_later +
                    m.global_prop.loc[(max(sorted_stf), 'Weight'), 'value
↪'] -
                    1 <= stf + m.process_dict['depreciation'][
                                        (stf, sit, pro)]):
                    op_pro.append((sit, pro, stf, stf_later))
            elif (sorted_stf[index_helper+1] <=
                  stf + m.process_dict['depreciation'][(stf, sit, pro)] and
                  stf <= stf_later):
                op_pro.append((sit, pro, stf, stf_later))
            else:
                pass

    return op_pro
```

Transmissions installed in an earlier support timeframe *stf* and still usable in support timeframe *stf_later* are collected in the following tuple set:

```
m.operational_tra_tuples = pyomo.Set(
    within=m.sit*m.sit*m.tra*m.com*m.stf*m.stf,
    initialize=[(sit, sit_, tra, com, stf, stf_later)
                for (sit, sit_, tra, com, stf, stf_later)
                in op_tra_tuples(m.tra_tuples, m)],
    doc='Transmissions that are still operational through stf_later'
        '(and the relevant years following), if built in stf'
    'in stf.')
```

where the following function is utilized:

```
def op_tra_tuples(tra_tuple, m):
    """ s.a. op_pro_tuples
    """
    op_tra = []
    sorted_stf = sorted(list(m.stf))

    for (stf, sit1, sit2, tra, com) in tra_tuple:
        for stf_later in sorted_stf:
            index_helper = sorted_stf.index(stf_later)
            if stf_later == max(sorted_stf):
                if (stf_later +
                    m.global_prop_dict['value'][(max(sorted_stf), 'Weight
↪')] -
                    1 <= stf + m.transmission_dict['depreciation'][
                        (stf, sit1, sit2, tra, com)]):
                    op_tra.append((sit1, sit2, tra, com, stf, stf_later))
            elif (sorted_stf[index_helper + 1] <=
                  stf + m.transmission_dict['depreciation'][
                      (stf, sit1, sit2, tra, com)] and stf <= stf_later):
                op_tra.append((sit1, sit2, tra, com, stf, stf_later))
            else:
                pass

    return op_tra
```

Storages installed in an earlier support timeframe *stf* and still usable in support timeframe *stf_later* are collected in the following tuple set:

```
m.operational_sto_tuples = pyomo.Set(
    within=m.sit*m.sto*m.com*m.stf*m.stf,
    initialize=[(sit, sto, com, stf, stf_later)
                for (sit, sto, com, stf, stf_later)
                in op_sto_tuples(m.sto_tuples, m)],
    doc='Processes that are still operational through stf_later'
        '(and the relevant years following), if built in stf'
    'in stf.')
```

where the following function is utilized:

```
def op_sto_tuples(sto_tuple, m):
    """ s.a. op_pro_tuples
    """
    op_sto = []
```

(continues on next page)

(continued from previous page)

```
    sorted_stf = sorted(list(m.stf))

    for (stf, sit, sto, com) in sto_tuple:
        for stf_later in sorted_stf:
            index_helper = sorted_stf.index(stf_later)
            if stf_later == max(sorted_stf):
                if (stf_later +
                    m.global_prop_dict['value'][(max(sorted_stf), 'Weight
↪')] -
                    1 <= stf +
                        m.storage_dict['depreciation'][(stf, sit, sto,␣
↪com)]):
                    op_sto.append((sit, sto, com, stf, stf_later))
            elif (sorted_stf[index_helper + 1] <=
                  stf +
                  m.storage_dict['depreciation'][(stf, sit, sto, com)] and
                  stf <= stf_later):
                op_sto.append((sit, sto, com, stf, stf_later))
            else:
                pass

    return op_sto
```

Variables

All the variables that the optimization model requires to calculate an optimal solution will be listed and defined in this section. A variable is a numerical value that is determined during optimization. Variables can denote a single, independent value, or an array of values. Variables define the search space for optimization. Variables of this optimization model can be separated into sections by their area of use. These Sections are Cost, Commodity, Process, Transmission, Storage and demand side management.

Table 4: *Table: Model Variables*

Variable	Unit	Description
Cost Variables		
ζ	€	Total System Cost
ζ_{inv}	€	Investment Costs
ζ_{fix}	€	Fix Costs
ζ_{var}	€	Variable Costs
ζ_{fuel}	€	Fuel Costs
ζ_{rev}	€	Revenue Costs
ζ_{pur}	€	Purchase Costs
ζ_{start}	€	Start Costs
Commodity Variables		
ρ_{yvct}	MWh	Stock Commodity Source Term
ϱ_{yvct}	MWh	Sell Commodity Source Term
ψ_{yvct}	MWh	Buy Commodity Source Term
Process Variables		
κ_{yvp}	MW	Total Process Capacity
$\hat{\kappa}_{yvp}$	MW	New Process Capacity

Continued on next page

Table 4 – continued from previous page

Variable	Unit	Description
β_{yvp}	•	New Process Capacity Units
τ_{yvpt}	MWh	Process Throughput
$\epsilon^{\text{in}}_{yvcpt}$	MWh	Process Input Commodity Flow
$\epsilon^{\text{out}}_{yvcpt}$	MWh	Process Output Commodity Flow
$yvpt$	•	Process On/Off Marker
σ_{yvpt}	•	Process Start-up Marker
Transmission Variables		
κ_{yaf}	MW	Total transmission Capacity
$\hat{\kappa}_{yaf}$	MW	New Transmission Capacity
β_{yaf}	•	New Transmission Capacity Units
π^{in}_{yaft}	MWh	Transmission Input Commodity Flow
π^{out}_{yaft}	MWh	Transmission Output Commodity Flow
DCPF Transmission Variables		
θ_{yvt}	deg.	Voltage Angle
$\pi^{\text{in}\prime}_{yaft}$	MW	Absolute Transmission Flow
Storage Variables		
κ^{c}_{yvs}	MWh	Total Storage Size
$\hat{\kappa}^{\text{c}}_{yvs}$	MWh	New Storage Size
β^{c}_{yvs}	•	New Storage Size Units
κ^{p}_{yvs}	MW	Total Storage Power
$\hat{\kappa}^{\text{p}}_{yvs}$	MW	New Storage Power
β^{c}_{yvs}	•	New Storage Power Units
$\epsilon^{\text{in}}_{yvst}$	MWh	Storage Input Commodity Flow
$\epsilon^{\text{out}}_{yvst}$	MWh	Storage Output Commodity Flow
$\epsilon^{\text{con}}_{yvst}$	MWh	Storage Energy Content
Demand Side Management Variables		
$\delta^{\text{up}}_{yvct}$	MWh	DSM Upshift
$\delta^{\text{down}}_{t,tt,yvc}$	MWh	DSM Downshift

Cost Variables

Total System Cost, ζ : the variable ζ represents the *total expense incurred* in reaching the satisfaction of the given energy demand in the entire modeling horizon. If only a fraction of a year is modeled in each support timeframe, the costs are scaled to the annual expenditures. The total cost is calculated by the sum total of all costs by type(ζ_r, $\forall r \in R$) and defined as `costs` by the following code fragment:

```
m.costs = pyomo.Var(
    m.cost_type,
    within=pyomo.Reals,
    doc='Costs by type (EUR/a)')
```

System costs are divided into the 7 cost types invest, fix, variable, fuel, purchase, sell and environmental. The separation of costs by type, facilitates business planning and provides calculation accuracy. These cost types are hardcoded, which means they are not considered to be fixed or changed by the user.

For more information on the definition of these variables see section *Minimal optimization model* and for their implementation see section *Objective function*.

Commodity Variables

Stock Commodity Source Term, ρ_{yvct}, `e_co_stock`, MWh : The variable ρ_{yvct} represents the energy amount in [MWh] that is being used by the system of commodity c from type stock ($\forall c \in C_{\text{stock}}$) in support timeframe y ($\forall y \in Y$) in a site v ($\forall v \in V$) at timestep t ($\forall t \in T_{\text{m}}$). In script `model.py` this variable is defined by the variable `e_co_stock` and initialized by the following code fragment:

```
m.e_co_stock = pyomo.Var(
    m.tm, m.com_tuples,
    within=pyomo.NonNegativeReals,
    doc='Use of stock commodity source (MWh) at a given timestep')
```

Sell Commodity Source Term, ϱ_{yvct}, `e_co_sell`, MWh : The variable ϱ_{yvct} represents the energy amount in [MWh] that is being used by the system of commodity c from type sell ($\forall c \in C_{\text{sell}}$) in support timeframe y ($\forall y \in Y$) in a site v ($\forall v \in V$) at timestep t ($\forall t \in T_{\text{m}}$). In script `model.py` this variable is defined by the variable `e_co_sell` and initialized by the following code fragment:

```
m.e_co_sell = pyomo.Var(
    m.tm, m.com_tuples,
    within=pyomo.NonNegativeReals,
    doc='Use of sell commodity source (MWh) at a given timestep')
```

Buy Commodity Source Term, ψ_{yvct}, `e_co_buy`, MWh : The variable ψ_{yvct} represents the energy amount in [MWh] that is being used by the system of commodity c from type buy ($\forall c \in C_{\text{buy}}$) in support timeframe y ($\forall y \in Y$) in a site v ($\forall v \in V$) at timestep t ($\forall t \in T_{\text{m}}$). In script `model.py` this variable is defined by the variable `e_co_buy` and initialized by the following code fragment:

```
m.e_co_buy = pyomo.Var(
   m.tm, m.com_tuples,
   within=pyomo.NonNegativeReals,
   doc='Use of buy commodity source (MWh) at a given timestep')
```

Process Variables

Total Process Capacity, κ_{yvp}, `cap_pro`: The variable κ_{yvp} represents the total potential throughput (capacity) of a process tuple p_{yv} ($\forall p \in P, \forall v \in V$, forall y in Y‘), that is required in the energy system. The total process capacity includes both the already installed process capacity and the additional new process capacity that needs to be installed. Since the costs of the process technologies are mostly directly proportional to the maximum possible output (and correspondingly to the capacity) of processes, this variable acts as a scale factor of process technologies. For further information see Process Capacity Rule. This variable is expressed in the unit (MW). In script `model.py` this variable is defined by the model variable `cap_pro` and initialized by the following code fragment:

```
m.cap_pro = pyomo.Var(
    m.pro_tuples,
    within=pyomo.NonNegativeReals,
    doc='Total process capacity (MW)')
```

New Process Capacity, $\hat{\kappa}_{yvp}$, `cap_pro_new`: The variable $\hat{\kappa}_{yvp}$ represents the capacity of a process tuple p_{yv} ($\forall p \in P, \forall v \in V$) that needs to be installed additionally to the energy system in support timeframe y in site v in order to provide the optimal solution. This variable is expressed in the unit MW. In script `model.py` this variable is defined by the model variable `cap_pro_new` and initialized by the following code fragment:

```
m.cap_pro_new = pyomo.Var(
    m.pro_tuples,
    within=pyomo.NonNegativeReals,
    doc='New process capacity (MW)')
```

New Process Capacity Units, β_{yvp}, `pro_cap_unit`: The variable β_{yvp} represents the number of capacity blocks of a process tuple p_{yv} ($\forall p \in P, \forall v \in V$) that needs to be installed additionally to the energy system in support timeframe y in site v in order to provide the optimal solution. In script `model.py` this variable is defined by the model variable `cap_pro_new` and initialized by the following code fragment:

```
m.pro_cap_unit = pyomo.Var(
    m.pro_tuples,
    within=pyomo.NonNegativeIntegers,
    doc='Number of newly installed capacity units')
```

Process Throughput, τ_{yvpt}, `tau_pro` : The variable τ_{yvpt} represents the measure of (energetic) activity of a process tuple p_{yv} ($\forall p \in P, \forall v \in V, \forall y \in Y$) at a timestep t ($\forall t \in T_m$). Based on the process throughput amount in a given timestep of a process, flow amounts of the process' input and output commodities at that timestep can be calculated by scaling the process throughput with corresponding process input and output ratios. For further information see **Process Input Ratio** and **Process Output Ratio**. The process throughput variable is expressed in the unit MWh. In script `model.py` this variable is defined by the model variable `tau_pro` and initialized by the following code fragment:

```
m.tau_pro = pyomo.Var(
    m.tm, m.pro_tuples,
    within=pyomo.NonNegativeReals,
    doc='Activity (MWh) through process')
```

Process Input Commodity Flow, $\epsilon^{\text{in}}_{yvcpt}$, `e_pro_in`: The variable $\epsilon^{\text{in}}_{yvcpt}$ represents the commodity input flow into a process tuple p_{yv} ($\forall p \in P, \forall v \in V, \forall y \in Y$) caused by an input commodity c ($\forall c \in C$)

at a timestep t ($\forall t \in T_m$). This variable is generally expressed in the unit MWh. In script `model.py` this variable is defined by the model variable `e_pro_in` and initialized by the following code fragment:

```
m.e_pro_in = pyomo.Var(
    m.tm, m.pro_tuples, m.com,
    within=pyomo.NonNegativeReals,
    doc='Flow of commodity into process at a given timestep')
```

Process Output Commodity Flow, $\epsilon^{\text{out}}_{yvcpt}$, `e_pro_out`: The variable $\epsilon^{\text{out}}_{vcpt}$ represents the commodity flow output out of a process tuple p_{yv} ($\forall p \in P, \forall v \in V, \forall y \in Y$) caused by an output commodity c ($\forall c \in C$) at a timestep t ($\forall t \in T_m$). This variable is generally expressed in the unit MWh (or tonnes e.g. for the environmental commodity 'CO2'). In script `model.py` this variable is defined by the model variable `e_pro_out` and initialized by the following code fragment:

```
m.e_pro_out = pyomo.Var(
    m.tm, m.pro_tuples, m.com,
    within=pyomo.NonNegativeReals,
    doc='Flow of commodity out of process at a given timestep')
```

Process On/Off Marker, $_{yvpt}$, `on_off`: The boolean variable $_{yvpt}$ marks whether process tuple p_{yv} ($\forall p \in P^{\text{on/off}}, \forall v \in V, \forall y \in Y$) is on and producing ($_{yvpt}$ is 1) or it is not producing ($_{yvpt}$ is 0) at a timestep t. While not producing, the process is either turned off or it started, without reaching the minimum fraction $\underline{P}_{yvp}$. In the script `AdvancedProcesses.py`, this variable is defined by the model variable `on_off` and initialized by the following code fragment:

```
m.on_off = pyomo.Var(
    m.t, m.pro_on_off_tuples,
    within=pyomo.Boolean,
    doc='Turn on/off a process with minimum working load')
```

Process Start-up Marker, σ_{yvpt}, `start_ups`: The boolean variable σ_{yvpt} marks whether process tuple p_{yv} ($\forall p \in P^{\text{on/off}}, \forall v \in V, \forall y \in Y$) is starting ($\sigma_{yvpt}$ becomes 1) or not (σ_{yvpt} is 0) at a timestep t. The process is considered to start when its output `e_pro_out` becomes greater than 0. In the script `AdvancedProcesses.py`, this variable is defined by the model variable `start_ups` and initialized by the following code fragment:

```
m.start_up = pyomo.Var(
        m.tm, m.pro_start_up_tuples,
        within=pyomo.Boolean,
        doc='Start-up marker')
```

Transmission Variables

Total Transmission Capacity, κ_{yaf}, `cap_tra`: The variable κ_{yaf} represents the total potential transfer power of a transmission tuple f_{yca}, where a represents the arc from an origin site v_{out} to a destination site v_{in}. The total transmission capacity includes both the already installed transmission capacity and the additional new transmission capacity that needs to be installed. This variable is expressed in the unit MW. In script `transmission.py` this variable is defined by the model variable `cap_tra` and initialized by the following code fragment:

```
m.cap_tra = pyomo.Var(
    m.tra_tuples,
```

(continues on next page)

(continued from previous page)

```
    within=pyomo.NonNegativeReals,
    doc='Total transmission capacity (MW)')
```

New Transmission Capacity, $\hat{\kappa}_{yaf}$, `cap_tra_new`: The variable $\hat{\kappa}_{yaf}$ represents the additional capacity, that needs to be installed, of a transmission tuple f_{yca}, where a represents the arc from an origin site v_{out} to a destination site v_{in}. This variable is expressed in the unit MW. In script `transmission.py` this variable is defined by the model variable `cap_tra_new` and initialized by the following code fragment:

```
m.cap_tra_new = pyomo.Var(
    m.tra_tuples,
    within=pyomo.NonNegativeReals,
    doc='New transmission capacity (MW)')
```

New Transmission Capacity Units, β_{yaf}, `tra_cap_unit`: The variable β_{yaf} represents the number of additional capacity blocks of a transmission tuple f_{yca} that need to be installed , where a represents the arc from an origin site v_{out} to a destination site v_{in}. In script `transmission.py` this variable is defined by the model variable `cap_tra_new` and initialized by the following code fragment:

```
m.tra_cap_unit =pyomo.Var(
    m.tra_block_tuples,
    within=pyomo.NonNegativeIntegers,
    doc='New transmission capacity blocks')
```

Transmission Input Commodity Flow, π^{in}_{yaft}, `e_tra_in`: The variable π^{in}_{yaft} represents the commodity flow input into a transmission tuple f_{yca} at a timestep t, where a represents the arc from an origin site v_{out} to a destination site v_{in}. This variable is expressed in the unit MWh. In script `urbs.py` this variable is defined by the model variable `e_tra_in` and initialized by the following code fragment:

```
m.e_tra_in = pyomo.Var(
    m.tm, m.tra_tuples,
    within=pyomo.NonNegativeReals,
    doc='Commodity flow into transmission line (MWh) at a given timestep')
```

Transmission Output Commodity Flow, π^{out}_{yaft}, `e_tra_out`: The variable π^{out}_{yaft} represents the commodity flow output out of a transmission tuple f_{ca} at a timestep t, where a represents the arc from an origin site v_{out} to a destination site v_{in}. This variable is expressed in the unit MWh. In script `urbs.py` this variable is defined by the model variable `e_tra_out` and initialized by the following code fragment:

```
m.e_tra_out = pyomo.Var(
    m.tm, m.tra_tuples,
    within=pyomo.NonNegativeReals,
    doc='Power flow out of transmission line (MWh) at a given timestep')
```

DCPF Transmission Variables

If the DC Power Flow transmission modelling is activated, two new variables are introduced to the model.

Voltage Angle, θ_{yvt}, `voltage_angle`: The variable θ_{yvt} represents the voltage angle of a site v, which has a DCPF transmission line connection, at a timestep t. This variable is expressed in the unit

degrees. In script `urbs.py` this variable is defined by the model variable `voltage_angle` and initialized by the following code fragment:

```
m.voltage_angle = pyomo.Var(
        m.tm, m.stf, m.sit,
        within=pyomo.Reals,
        doc='Voltage angle of a site')
```

Absolute Value of Transmission Commodity Flow, $\pi^{\text{in}}_{yaft}{}'$, `e_tra_abs`: The variable $\pi^{\text{in}}_{yaft}{}'$ represents the absolute value of the transmission commodity flow on a DCPF transmission tuple f_{yca} at a timestep t, where a represents the arc from an origin site v_{out} to a destination site v_{in}. This variable is expressed in the unit MWh. In script `urbs.py` this variable is defined by the model variable `e_tra_abs` and initialized by the following code fragment:

```
m.e_tra_abs = pyomo.Var(
    m.tm, m.tra_tuples_dc,
    within=pyomo.NonNegativeReals,
    doc='Absolute power flow on transmission line (MW) per timestep')
```

Transmission Commodity Flow Domain Changes :DC Power Flow transmission lines are represented by bidirectional single arcs instead of unidirectional symmetrical arcs as in the default transmission model. Consequently the power flow is allowed to be both positive or negative for DCPF transmission lines contrary to the transport transmission lines. For this reason, the domains of the variables transmission input commodity flow π^{in}_{yaft} and transmission output commodity flow π^{out}_{yaft} are defined with the `e_tra_domain_rule()` function depending on the corresponding transmission tuple set. These variables are defined by the model variables `e_tra_in` and `e_tra_out` and intialized by the code fragment:

```
m.e_tra_in = pyomo.Var(
    m.tm, m.tra_tuples,
    within=e_tra_domain_rule,
    doc='Power flow into transmission line (MW) per timestep')
m.e_tra_out = pyomo.Var(
    m.tm, m.tra_tuples,
    within=e_tra_domain_rule,
    doc='Power flow out of transmission line (MW) per timestep')
```

The function `e_tra_domain_rule()` is given by the code fragment:

```
def e_tra_domain_rule(m, tm, stf, sin, sout, tra, com):
    # assigning e_tra_in and e_tra_out variable domains for transport and␣
↪DCPF
    if (stf, sin, sout, tra, com) in m.tra_tuples_dc:
        return pyomo.Reals
    elif (stf, sin, sout, tra, com) in m.tra_tuples_tp:
        return pyomo.NonNegativeReals
```

Storage Variables

Total Storage Size, κ^{c}_{yvs}, `cap_sto_c`: The variable κ^{c}_{yvs} represents the total load capacity of a storage tuple s_{yvc}. The total storage load capacity includes both the already installed storage load capacity and the additional new storage load capacity that needs to be installed. This variable is expressed in unit MWh. In script `model.py` this variable is defined by the model variable `cap_sto_c` and initialized by the following code fragment:

```
m.cap_sto_c = pyomo.Var(
    m.sto_tuples,
    within=pyomo.NonNegativeReals,
    doc='Total storage size (MWh)')
```

New Storage Size, $\hat{\kappa}^{c}_{yvs}$, `cap_sto_c_new`: The variable $\hat{\kappa}^{c}_{yvs}$ represents the additional storage load capacity of a storage tuple s_{vc} that needs to be installed to the energy system in order to provide the optimal solution. This variable is expressed in the unit MWh. In script `model.py` this variable is defined by the model variable `cap_sto_c_new` and initialized by the following code fragment:

```
m.cap_sto_c_new = pyomo.Var(
    m.sto_tuples,
    within=pyomo.NonNegativeReals,
    doc='New storage size (MWh)')
```

New Storage Size Units, β^{c}_{yvs}, `sto_cap_c_unit`: The variable $\hat{\kappa}^{c}_{yvs}$ represents the number of additional storage load capacity blocks of a storage tuple s_{vc} that needs to be installed to the energy system in order to provide the optimal solution. In script `storage.py` this variable is defined by the model variable `cap_sto_c_unit` and initialized by the following code fragment:

```
m.sto_cap_c_unit = pyomo.Var(
    m.sto_block_c_tuples,
    within=pyomo.NonNegativeIntegers,
    doc='New storage size units')
```

Total Storage Power, κ^{p}_{yvs}, `cap_sto_p`: The variable κ^{p}_{yvs} represents the total potential discharge power of a storage tuple s_{vc}. The total storage power includes both the already installed storage power and the additional new storage power that needs to be installed. This variable is expressed in the unit MW. In script `model.py` this variable is defined by the model variable `cap_sto_p` and initialized by the following code fragment:

```
m.cap_sto_p = pyomo.Var(
    m.sto_tuples,
    within=pyomo.NonNegativeReals,
    doc='Total storage power (MW)')
```

New Storage Power, $\hat{\kappa}^{p}_{yvs}$, `cap_sto_p_new`: The variable $\hat{\kappa}^{p}_{yvs}$ represents the additional potential discharge power of a storage tuple s_{vc} that needs to be installed to the energy system in order to provide the optimal solution. This variable is expressed in the unit MW. In script `model.py` this variable is defined by the model variable `cap_sto_p_new` and initialized by the following code fragment:

```
m.cap_sto_p_new = pyomo.Var(
    m.sto_tuples,
    within=pyomo.NonNegativeReals,
    doc='New  storage power (MW)')
```

New Storage Power Units, β^{c}_{yvs}, `sto_cap_p_unit`: The variable β^{c}_{yvs} represents the number of additional potential discharge power blocks of a storage tuple s_{vc} that needs to be installed to the energy system in order to provide the optimal solution. In the script `storage.py` this variable is defined by the model variable `sto_cap_p_unit` and initialized by the following code fragment:

```
m.sto_cap_p_unit = pyomo.Var(
    m.sto_block_p_tuples,
```

(continues on next page)

(continued from previous page)

```
    within=pyomo.NonNegativeIntegers,
    doc='New storage power units')
```

Storage Input Commodity Flow, $\epsilon^{\text{in}}_{yvst}$, `e_sto_in`: The variable $\epsilon^{\text{in}}_{yvst}$ represents the input commodity flow into a storage tuple s_{yvc} at a timestep t. Input commodity flow into a storage tuple can also be defined as the charge of a storage tuple. This variable is expressed in the unit MWh. In script `model.py` this variable is defined by the model variable `e_sto_in` and initialized by the following code fragment:

```
m.e_sto_in = pyomo.Var(
    m.tm, m.sto_tuples,
    within=pyomo.NonNegativeReals,
    doc='Commodity flow into storage (MWh) at a given timestep')
```

Storage Output Commodity Flow, $\epsilon^{\text{out}}_{yvst}$, `e_sto_out`: The variable $\epsilon^{\text{out}}_{vst}$ represents the output commodity flow out of a storage tuple s_{yvc} at a timestep t. Output commodity flow out of a storage tuple can also be defined as the discharge of a storage tuple. This variable is expressed in the unit MWh. In script `model.py` this variable is defined by the model variable `e_sto_out` and initialized by the following code fragment:

```
m.e_sto_out = pyomo.Var(
    m.tm, m.sto_tuples,
    within=pyomo.NonNegativeReals,
    doc='Commodity flow out of storage (MWh) at a given timestep')
```

Storage Energy Content, $\epsilon^{\text{con}}_{yvst}$, `e_sto_con`: The variable $\epsilon^{\text{con}}_{yvst}$ represents the energy amount that is loaded in a storage tuple s_{vc} at a timestep t. This variable is expressed in the unit MWh. In script `urbs.py` this variable is defined by the model variable `e_sto_out` and initialized by the following code fragment:

```
m.e_sto_con = pyomo.Var(
    m.t, m.sto_tuples,
    within=pyomo.NonNegativeReals,
    doc='Energy content of storage (MWh) at a given timestep')
```

Demand Side Management Variables

DSM Upshift, $\delta^{\text{up}}_{yvct}$, `dsm_up`, MWh: The variable $\delta^{\text{up}}_{yvct}$ represents the DSM upshift in time step t in support timeframe y in site v for commodity c. It is only defined for all `dsm_site_tuples`. The following code fragment shows the definition of the variable:

```
m.dsm_up = pyomo.Var(
    m.tm, m.dsm_site_tuples,
    within=pyomo.NonNegativeReals,
    doc='DSM upshift (MWh) of a demand commodity at a given timestap')
```

DSM Downshift, $\delta^{\text{down}}_{t,tt,yvc}$, `dsm_down`, MWh: The variable $\delta^{\text{down}}_{t,tt,yvc}$ represents the DSM downshift in timestep tt caused by the upshift in time t in support timeframe y in site v for commodity c. The special combinations of timesteps t and tt for each (support timeframe, site, commodity) combination is created by the `dsm_down_tuples`. The definition of the variable is shown in the code fragment:

```
m.dsm_down = pyomo.Var(
m.dsm_down_tuples,
within=pyomo.NonNegativeReals,
doc='DSM downshift (MWh) of a demand commodity at a given timestep')
```

Parameters

All the parameters that the optimization model requires to calculate an optimal solution will be listed and defined in this section. A parameter is a datapoint, that is provided by the user before the optimization simulation starts. These parameters are the values that define the specifications of the modelled energy system. Parameters of this optimization model can be separated into two main parts, these are Technical and Economical Parameters.

Technical Parameters

Table 5: *Table: Technical Model Parameters*

Parameter	Unit	Description
General Technical Parameters		
w	_	Fraction of 1 year of modeled timesteps
Δt	h	Timestep Size
W	a	Weight of last support timeframe
Commodity Technical Parameters		
d_{yvct}	MWh	Demand for Commodity
s_{yvct}	_	Intermittent Supply Capacity Factor
$\overline{l}_{yvc}$	MW	Maximum Stock Supply Limit Per Hour
$\overline{L}_{yvc}$	MWh	Maximum Annual Stock Supply Limit Per Vertex
$\overline{m}_{yvc}$	t/h	Maximum Environmental Output Per Hour
$\overline{M}_{yvc}$	t	Maximum Annual Environmental Output
$\overline{g}_{yvc}$	MW	Maximum Sell Limit Per Hour
$\overline{G}_{yvc}$	MWh	Maximum Annual Sell Limit
$\overline{b}_{yvc}$	MW	Maximum Buy Limit Per Hour
$\overline{B}_{yvc}$	MWh	Maximum Annual Buy Limit
$\overline{L}_{\mathrm{CO_2},y}$	t	Maximum Global Annual CO2 Emission Limit
$\overline{\overline{L}}_{\mathrm{CO_2}}$	t	CO2 Emission Budget for modeling horizon
Process Technical Parameters		
$\underline{K}_{yvp}$	MW	Process Capacity Lower Bound
K_{vp}	MW	Process Capacity Installed
$\overline{K}_{yvp}$	MW	Process Capacity Upper Bound
T_{vp}	MW	Remaining lifetime of installed processes
$\overline{PG}^{\mathrm{up}}_{yvp}$	1/h	Process Maximal Power Ramp Up Gradient (relat
$\overline{PG}^{\mathrm{down}}_{yvp}$	1/h	Process Maximal Power Ramp Down Gradient (r
$\overline{ST}_{yvp}$	h	Process Starting Time
$\overline{SR}_{yvp}$	1/h	Process Starting Ramp
$\underline{P}_{yvp}$	_	Process Minimum Part Load Fraction
f^{out}_{yvpt}	_	Process Output Ratio multiplyer
r^{in}_{ypc}	_	Process Input Ratio

Continued on nex

Table 5 – continued from previous page

Parameter	Unit	Description
$\underline{r}^{\text{in}}_{ypc}$	_	Process Partial Input Ratio
$\underline{r}^{\text{out}}_{ypc}$	_	Process Partial Output Ratio
r^{out}_{ypc}	_	Process Output Ratio
K^{block}_{yvp}	MW	Process New Capacity Block
Storage Technical Parameters		
I_{yvs}	_	Initial and Final State of Charge
e^{in}_{yvs}	_	Storage Efficiency During Charge
e^{out}_{yvs}	_	Storage Efficiency During Discharge
d_{yvs}	1/h	Storage Self-discharge Per Hour
$\underline{K}^{\text{c}}_{yvs}$	MWh	Storage Capacity Lower Bound
K^{c}_{yvs}	MWh	Storage Capacity Installed
$\overline{K}^{\text{c}}_{yvs}$	MWh	Storage Capacity Upper Bound
$\underline{K}^{\text{p}}_{yvs}$	MW	Storage Power Lower Bound
K^{p}_{yvs}	MW	Storage Power Installed
$\overline{K}^{\text{p}}_{yvs}$	MW	Storage Power Upper Bound
T_{vs}	MW	Remaining lifetime of installed storages
$k^{\text{E/P}}_{yvs}$	h	Storage Energy to Power Ratio
$K^{\text{c,block}}_{yvs}$	MWh	Storage New Capacity Block
$K^{\text{p,block}}_{yvs}$	MW	Storage New Power Block
Transmission Technical Parameters		
e_{yaf}	_	Transmission Efficiency
$\underline{K}_{yaf}$	MW	Transmission Capacity Lower Bound
K_{yaf}	MW	Transmission Capacity Installed
$\overline{K}_{yaf}$	MW	Transmission Capacity Upper Bound
T_{af}	year	Remaining lifetime of installed transmission
K^{block}_{yaf}	MW	Transmission New Capacity Block
DCPF Transmission Technical Parameters		
X_{yaf}	p.u	Transmission Reactance
$\overline{dl}_{yaf}$	deg.	Voltage Angle Difference Limit
$V_{yaf\text{base}}$	kV	Transmission Base Voltage
K^{block}_{yaf}	_	Transmission New Capacity Block
Demand Side Management Parameters		
e_{yvc}	_	DSM Efficiency
y_{yvc}	_	DSM Delay Time
o_{yvc}	_	DSM Recovery Time
$\overline{K}^{\text{up}}_{yvc}$	MW	DSM Maximal Upshift Per Hour
$\overline{K}^{\text{down}}_{yvc}$	MW	DSM Maximal Downshift Per Hour

General Technical Parameters

Weight, w, `weight`: The parameter w helps to scale variable costs and emissions from the length of simulation, that the energy system model is being observed, to an annual result. This parameter represents the fraction of a year (8760 hours) of the observed time span. The observed time span is calculated by the product of number of time steps of the set T and the time step duration. In script `model.py` this parameter is defined by the model parameter `weight` and initialized by the following code fragment:

```
m.weight = pyomo.Param(
    initialize=float(8760) / (len(m.tm) * dt),
    doc='Pre-factor for variable costs and emissions for an annual result')
```

Timestep Duration, Δt, `dt`: The parameter Δt represents the duration between two sequential timesteps t_x and t_{x+1}. This is calculated by the subtraction of smaller one from the bigger of the two sequential timesteps $\Delta t = t_{x+1} - t_x$. This parameter is the unit of time for the optimization model, is expressed in the unit h and by default the value is set to `1`. In script `model.py` this parameter is defined by the model parameter `dt` and initialized by the following code fragment:

```
m.dt = pyomo.Param(
    initialize=dt,
    doc='Time step duration (in hours), default: 1')
```

The user can set the paramteter in script `runme.py` in the line:

```
dt = 1  # length of each time step (unit: hours)
```

Weight of last modeled support timeframe, W, `m.global_prop.loc[(min(m.stf), 'Cost budget'), 'value']`: This parameter specifies how long the time interval represented by the last support timeframe is. The unit of this parameter is years. By extension it also specifies the end of the modeling horizon. The parameter is set in the spreadsheet corresponding to the last support timeframe in worksheet "Global" in the line denoted "Weight" in the column titled "value".

Commodity Technical Parameters

Demand for Commodity, d_{yvct}, `m.demand_dict[(stf, sit, com)][tm]`: The parameter represents the energy amount of a demand commodity tuple c_{yvq} required at a timestep t ($\forall y \in Y, \forall v \in V, q = "Demand", \forall t \in T_m$). The unit of this parameter is MWh. This data is to be provided by the user and to be entered in the spreadsheet corresponding to the specified support timeframe. The related section for this parameter in the spreadsheet can be found in the "Demand" sheet. Here each row represents another timestep t and each column represent a commodity tuple c_{yvq}. Rows are named after the timestep number n of timesteps t_n. Columns are named after the combination of site name v and commodity name c respecting the order and seperated by a period(.). For example (Mid, Elec) represents the commodity Elec in site Mid. Commodity Type q is omitted in column declarations, because every commodity of this parameter has to be from commodity type *Demand* in any case.

Intermittent Supply Capacity Factor, s_{yvct}, `m.supim_dict[(stf, sit, coin)][tm]`: The parameter s_{yvct} represents the normalized availability of a supply intermittent commodity c ($\forall c \in C_{\text{sup}}$) in a support timeframe y and site v at a timestep t. In other words this parameter gives the ratio of current available energy amount to maximum potential energy amount of a supply intermittent commodity. This data is to be provided by the user and to be entered in the spreadsheet corresponding to the support timeframe. The related section for this parameter in the spreadsheet can be found under the "SupIm" sheet. Here each row represents another timestep t and each column represent a commodity tuple c_{vq}. Rows are named after the timestep number n of timesteps t_n. Columns are named after the combination of site name v and commodity name c, in this respective order and separated by a period(.). For example (Mid.Elec) represents the commodity Elec in site Mid. Commodity Type q is omitted in column declarations, because every commodity of this parameter has to be from commodity type *SupIm* in any case.

Maximum Stock Supply Limit Per Hour, $\bar{l}_{yvc}$, `m.commodity_dict['maxperhour'][(stf, sit, com, com_type)]`: The parameter $\bar{l}_{yvc}$ represents the maximum energy amount of a stock

commodity tuple c_{yvq} ($\forall y \in Y, \forall v \in V, q =$ "$Stock$") that energy model is allowed to use per hour. The unit of this parameter is MW. This parameter applies to every timestep and does not vary for each timestep t. This parameter is to be provided by the user and to be entered in spreadsheet corresponding to the support timeframe. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{yvq} and the column with the header label "maxperhour" represents the parameter $\overline{l}_{yvc}$. If there is no desired restriction of a stock commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Annual Stock Supply Limit Per Vertex, $\overline{L}_{yvc}$, `m.commodity_dict['max'][(stf, sit, com, com_type)]`: The parameter $\overline{L}_{yvc}$ represents the maximum energy amount of a stock commodity tuple c_{yvq} ($\forall y \in Y, \forall v \in V, q =$ "$Stock$") that energy model is allowed to use annually. The unit of this parameter is MWh. This parameter is to be provided by the user and to be entered in spreadsheet corresponding to the support timeframe. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{yvq} and the column with the header label "max" represents the parameter $\overline{L}_{yvc}$. If there is no desired restriction of a stock commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Environmental Output Per Hour, $\overline{m}_{yvc}$, `m.commodity_dict['maxperhour'][(stf, sit, com, com_type)]`: The parameter $\overline{m}_{yvc}$ represents the maximum energy amount of an environmental commodity tuple c_{yvq} ($\forall y \in Y, \forall v \in V, q =$ "Env") that energy model is allowed to produce and release to environment per time step. This parameter applies to every timestep and does not vary for each timestep t/h. This parameter is to be provided by the user and to be entered in spreadsheet corresponding to the support timeframe. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{yvq} and the column with the header label "maxperhour" represents the parameter $\overline{m}_{yvc}$. If there is no desired restriction of an environmental commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Annual Environmental Output, $\overline{M}_{yvc}$, `m.commodity_dict['max'][(stf, sit, com, com_type)]`: The parameter $\overline{M}_{vc}$ represents the maximum energy amount of an environmental commodity tuple c_{yvq} ($\forall y \in Y, \forall v \in V, q =$ "Env") that energy model is allowed to produce and release to environment annually. This parameter is to be provided by the user and to be entered in spreadsheet corresponding to the support timeframe. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{yvq} and the column with the header label "max" represents the parameter $\overline{M}_{yvc}$. If there is no desired restriction of a stock commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Sell Limit Per Hour, $\overline{g}_{yvc}$, `m.commodity_dict['maxperhour'][(stf, sit, com, com_type)]`: The parameter $\overline{g}_{yvc}$ represents the maximum energy amount of a sell commodity tuple c_{yvq} ($\forall y \in Y, \forall v \in V, q =$ "$Sell$") that energy model is allowed to sell per hour. The unit of this parameter is MW. This parameter applies to every timestep and does not vary for each timestep t. This parameter is to be provided by the user and to be entered in spreadsheet. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{yvq} and the column with the header label "maxperhour" represents the parameter $\overline{g}_{yvc}$. If there is no desired restriction of a sell commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Annual Sell Limit, $\overline{G}_{yvc}$, `m.commodity_dict['max'][(stf, sit, com, com_type)]`: The parameter $\overline{G}_{yvc}$ represents the maximum energy amount of a sell commodity tuple c_{yvq} ($\forall y \in Y, \forall v \in V, q =$ "$Sell$") that energy model is allowed to sell annually. The unit of this parameter is MWh. This parameter is to be provided by the user and to be entered in spreadsheet corresponding to the support timeframe. The related section for this parameter in the spreadsheet can be found under

the `Commodity` sheet. Here each row represents another commodity tuple c_{yvq} and the column of sell with the header label "max" represents the parameter $\overline{G}_{yvc}$. If there is no desired restriction of a sell commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Buy Limit Per Hour, $\overline{b}_{yvc}$, `m.commodity_dict['maxperhour'][(stf, sit, com, com_type)]`: The parameter $\overline{b}_{yvc}$ represents the maximum energy amount of a buy commodity tuple c_{yvq} ($\forall y \in Y, \forall v \in V, q = "Buy"$) that energy model is allowed to buy per hour. The unit of this parameter is MW. This parameter applies to every timestep and does not vary for each timestep t. This parameter is to be provided by the user and to be entered in spreadsheet corresponding to the support timeframe. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{yvq} and the column with the header label "maxperhour" represents the parameter $\overline{b}_{yvc}$. If there is no desired restriction of a sell commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Annual Buy Limit, $\overline{B}_{yvc}$, `m.commodity_dict['max'][(stf, sit, com, com_type)]`: The parameter $\overline{B}_{yvc}$ represents the maximum energy amount of a buy commodity tuple c_{yvq} ($\forall y \in Y, \forall v \in V, q = "Buy"$) that energy model is allowed to buy annually. The unit of this parameter is MWh. This parameter is to be provided by the user and to be entered in spreadsheet corresponding to the support timeframe. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{yvq} and the column with the header label "max" represents the parameter $\overline{B}_{yvc}$. If there is no desired restriction of a buy commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Global Annual CO_2 Annual Emission Limit, $\overline{L}_{CO_2,y}$, `m.global_prop.loc[stf, 'CO2 limit']['value']`: The parameter $\overline{L}_{CO_2,y}$ represents the maximum total amount of CO2 the energy model is allowed to produce and release to the environment annually. If the user desires to set a maximum annual limit to total CO_2 emission across all sites of the energy model in a given support timeframe y, this can be done by entering the desired value to the spreadsheet corresponding to the support timeframe. The related section for this parameter can be found under the sheet "Global". Here the the cell where the "CO2 limit" row and "value" column intersects stands for the parameter $\overline{L}_{CO_2,y}$. If the user wants to disable this parameter and restriction it provides, this cell can be set to "inf" or simply be deleted.

CO_2overline{overline{L}}_{CO_2}', `m.global_prop.loc[min(m.stf), 'CO2 budget']['value']`: The parameter $\overline{\overline{L}}_{CO_2}$ represents the maximum total amount of CO2 the energy model is allowed to produce and release to the environment over the entire modeling horizon. If the user desires to set a limit to total CO_2 emission across all sites and the entire modeling horizon of the energy model, this can be done by entering the desired value to the spreadsheet of the first support timeframe. The related section for this parameter can be found under the sheet "Global". Here the the cell where the "CO2 budget" row and "value" column intersects stands for the parameter $\overline{\overline{L}}_{CO_2}$. If the user wants to disable this parameter and restriction it provides, this cell can be set to "inf" or simply be deleted.

Process Technical Parameters

Process Capacity Lower Bound, $\underline{K}_{yvp}$, `m.process_dict['cap-lo'][stf, sit, pro]`: The parameter $\underline{K}_{yvp}$ represents the minimum amount of power output capacity of a process p at a site v in support timeframe y, that energy model is required to have. The unit of this parameter is MW. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "cap-lo" represents the parameters $\underline{K}_{yvp}$ belonging to the corresponding process p

and site v combinations. If there is no desired minimum limit for the process capacities, this parameter can be simply set to "0".

Process Capacity Installed, K_{vp}, `m.process_dict['inst-cap'][min(m.stf), sit, pro]`: The parameter K_{vp} represents the amount of power output capacity of a process p in a site v, that is already installed to the energy system at the beginning of the modeling period. The unit of this parameter is MW. The related section for this parameter can be found in the spreadsheet corresponding to the first support timeframe under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "inst-cap" represents the parameters K_{vp} belonging to the corresponding process p and site v combinations.

Process Capacity Upper Bound, $\overline{K}_{yvp}$, `m.process_dict['cap-up'][stf, sit, pro]`: The parameter $\overline{K}_{yvp}$ represents the maximum amount of power output capacity of a process p at a site v in support timeframe y, that energy model is allowed to have. The unit of this parameter is MW. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "cap-up" represents the parameters $\overline{K}_{yvp}$ of the corresponding process p and site v combinations. Alternatively, $\overline{K}_{yvp}$ is determined by the column with the label "area-per-cap", whenever the value in "cap-up" times the value "area-per-cap" is larger than the value in column "area" in sheet "Site" for site v in support timeframe y. If there is no desired maximum limit for the process capacities, both input parameters can be simply set to "inf".

Remaining lifetime of installed processes, T_{vp}, `m.process.loc[(min(m.stf), sit, pro), 'lifetime']`: The parameter T_{vp} represents the remaining lifetime of already installed units. It is used to determine the set *m.inst_pro_tuples*, i.e. to identify for which support timeframes the installed unit can still be used.

Process Maximal Power Ramp Up Gradient, $\overline{PG}^{\text{up}}_{yvp}$, `m.process_dict['ramp-up-grad'][(stf, sit, pro)]`: The parameter $\overline{PG}^{\text{up}}_{yvp}$ represents the maximal power ramp up gradient of a process p at a site v in support timeframe y, that energy model is allowed to have. The unit of this parameter is 1/h. The related section for this parameter in the spreadsheet can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "ramp-up-grad" represents the parameters $\overline{PG}^{\text{up}}_{yvp}$ of the corresponding process p and site v combinations. If there is no desired maximum limit for the process power ramp up gradient, this parameter can be simply set to a value larger or equal to 1.

Process Maximal Power Ramp Down Gradient, $\overline{PG}^{\text{down}}_{yvp}$, `m.process_dict['ramp-down-grad'][(stf, sit, pro)]`: The parameter $\overline{PG}^{\text{down}}_{yvp}$ represents the maximal power ramp down gradient of a process p at a site v in support timeframe y, that energy model is allowed to have. The unit of this parameter is 1/h. The related section for this parameter in the spreadsheet can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "ramp-down-grad" represents the parameters $\overline{PG}^{\text{up}}_{yvp}$ of the corresponding process p and site v combinations. If there is no desired maximum limit for the process power ramp down gradient, this parameter can be simply set to a value larger or equal to 1.

Process Starting Time, $\overline{ST}_{yvp}$, `m.process_dict['start-time'][(stf, sit, pro)]`: The parameter $\overline{ST}_{yvp}$ represents the time required by a process p at a site v in support timeframe y to start. The unit of this parameter is h. The related section for this parameter in the spreadsheet can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "start-time" represents the parameters $\overline{ST}_{yvp}$ of the corresponding process p and site v combinations.

Process Starting Ramp, $\overline{SR}_{yvp}$: The parameter $\overline{SR}_{yvp}$ represents the ramp of a process p at a site v in

support timeframe y while starting. The unit of this parameter is 1/h. This parameter is not declared directly in the input, being only a derived parameter, calculated as the ratio between the process minimum part load fraction $\underline{P}_{yvp}$ and the process starting time $\overline{ST}_{yvp}$.

Process Minimum Part Load Fraction, $\underline{P}_{yvp}$, `m.process_dict['min-fraction'][(stf, sit, pro)]`: The parameter $\underline{P}_{yvp}$ represents the minimum allowable part load of a process p at a site v in support timeframe y as a fraction of the total process capacity. The related section for this parameter in the spreadsheet can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "min-fraction" represents the parameters $\underline{P}_{yvp}$ of the corresponding process p and site v combinations. The minimum part load fraction parameter constraints is only relevant when the part load behavior for the process is active, i.e., when in the process commodity sheet a value for "ratio-min" is set for at least one input commodity.

Process Output Ratio multiplyer, f^{out}_{yvpt}, `m.eff_factor_dict[(stf, sit, pro)]`: The parameter time series f^{out}_{yvpt} allows for a time dependent modification of process outputs and by extension of the efficiency of a process p in site v and support timeframe y. It can be used, e.g., to model temperature dependent efficiencies of processes or to include scheduled maintenance intervals. In the spreadsheet corresponding to the support timeframe this timeseries is set in worksheet "TimeVarEff". Here each row represents another timestep t and each column represent a process tuple p_{yv}. Rows are named after the timestep number n of timesteps t_n. Columns are named after the combination of site name v and commodity name and process name p respecting the order and seperated by a period(.). For example (Mid, Lignite plant) represents the process Lignite plant in site Mid. Note that the output of environmental commodity outputs are not manipulated by this factor as it is typically linked to an input commodity as , e.g., CO2 output is linked to a fossil input.

Process Input Ratio, r^{in}_{ypc}, `m.r_in_dict[(stf, pro, co)]`: The parameter r^{in}_{ypc} represents the ratio of the input amount of a commodity c in a process p and support timeframe y, relative to the process throughput at a given timestep. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Process-Commodity" sheet. Here each row represents another commodity c that either goes in to or comes out of a process p. The column with the header label "ratio" represents the parameters r^{in}_{ypc} of the corresponding process p and commodity c if the latter is an input commodity.

Process Partial Input Ratio, $\underline{r}^{\text{in}}_{ypc}$, `m.r_in_min_fraction[stf, pro, coin]`: The parameter $\underline{r}^{\text{in}}_{ypc}$ represents the ratio of the amount of input commodity c a process p and support timeframe y consumes if it is at its minimum allowable partial operation. More precisely, when its throughput τ_{yvpt} has the minimum value $\kappa_{yvp}\underline{P}_{yvp}$. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Process-Commodity" sheet. Here each row represents another commodity c that either goes in to or comes out of a process p. The column with the header label "ratio-min" represents the parameters $\underline{r}^{\text{in,out}}_{ypc}$ of the corresponding process p and commodity c if the latter is an input commodity.

Process Output Ratio, r^{out}_{ypc}, `m.r_out_dict[(stf, pro, co)]`: The parameter r^{out}_{ypc} represents the ratio of the output amount of a commodity c in a process p in support timeframe y, relative to the process throughput at a given timestep. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Process-Commodity" sheet. Here each row represents another commodity c that either goes in to or comes out of a process p. The column with the header label "ratio" represents the parameters of the corresponding process p and commodity c if the latter is an output commodity.

Process Partial Output Ratio, $\underline{r}^{\text{out}}_{ypc}$, `m.r_out_min_fraction[stf, pro, coo]`: The parameter $\underline{r}^{\text{out}}_{ypc}$ represents the ratio of the amount of output commodity c a process p and support timeframe y emits if it is at its minimum allowable partial operation. More precisely, when its throughput τ_{yvpt} has the minimum value $\kappa_{yvp}\underline{P}_{yvp}$. The related section for this parameter in the spreadsheet corresponding

to the support timeframe can be found under the "Process-Commodity" sheet. Here each row represents another commodity c that either goes in to or comes out of a process p. The column with the header label "ratio-min" represents the parameters $\underline{r}_{ypc}^{\text{in,out}}$ of the corresponding process p and commodity c if the latter is an output commodity.

Process input and output ratios are, in general, used for unit conversion between the different commodities.

Since all costs and capacity constraints take the process throughput τ_{yvpt} as the reference, it is reasonable to assign an in- or output ratio of "1" to at least one commodity. The flow of this commodity then tracks the throughput and can be used as a reference. All other values of in- and output ratios can then be adjusted by scaling them by an appropriate factor to the reference commodity flow.

Process New Capacity Block, K_{yvp}^{block}, `m.process_dict['cap-block'][(stf, sit, pro)]`: The parameter K_{yvp}^{block} represents the capacity of all newly installed units of a process p at a site v in the support timeframe y. The unit of this parameter is MW. The related section for this parameter in the spreadsheet can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "cap-block" represents the parameters K_{yvp}^{block} of the corresponding process p and site v combinations.

Storage Technical Parameters

Initial and Final State of Charge (relative), I_{yvs}, `m.storage_dict['init'][(stf, sit, sto, com)]`: The parameter I_{yvs} represents the initial state of charge of a storage s in a site v and support timeframe y. If this value is left unspecified, the initial state of charge is variable. The initial and final value are set as identical in each modeled support timeframe to avoid windfall profits through emptying of a storage. The value of this parameter is expressed as a normalized percentage, where "1" represents a fully loaded storage and "0" represents an empty storage. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "init" represents the parameters for corresponding storage s, site v, commodity c combinations. When no initial value is to be set this cell can be left empty.

Storage Efficiency During Charge, e_{yvs}^{in}, `m.storage_dict['eff-in'][(stf, sit, sto, com)]`: The parameter e_{yvs}^{in} represents the charging efficiency of a storage s in a site v and support timeframe y that stores a commodity c. The charging efficiency shows, how much of a desired energy and accordingly power can be successfully stored into a storage. The value of this parameter is expressed as a normalized percentage, where "1" represents a charging without energy losses. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "eff-in" represents the parameters e_{yvs}^{in} for corresponding storage tuples.

Storage Efficiency During discharge, e_{yvs}^{out}, `m.storage_dict['eff-out'][(stf, sit, sto, com)]`: The parameter e_{yvs}^{out} represents the discharging efficiency of a storage s in a site v and support timeframe y that stores a commodity c. The discharging efficiency shows, how much of a desired energy and accordingly power can be successfully released from a storage. The value of this parameter is expressed as a normalized percentage, where "1" represents a discharging without energy losses. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "eff-out" represents the parameters e_{yvs}^{out} for corresponding storage tuples.

Storage Self-discharge Per Hour, d_{yvs}, `m.storage_dict['discharge'][(stf, sit, sto, com)]`: The parameter d_{vs} represents the fraction of the energy content within a storage which is lost due to self-discharge per hour. It introduces an exponential decay of a given storage state if no charging/discharging takes place. The unit of this parameter is 1/h. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "discharge" represents the parameters d_{yvs} for corresponding storage tuples.

Storage Capacity Lower Bound, $\underline{K}^{\mathrm{c}}_{yvs}$, `m.storage_dict['cap-lo-c'][(stf, sit, sto, com)]`: The parameter $\underline{K}^{\mathrm{c}}_{yvs}$ represents the minimum amount of energy content capacity required for a storage s storing a commodity c in a site v in support timeframe y. The unit of this parameter is MWh. The related section for this parameter in the spreadsheet can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "cap-lo-c" represents the parameters $\underline{K}^{\mathrm{c}}_{yvs}$ for corresponding storage tuples. If there is no desired minimum limit for the storage energy content capacities, this parameter can be simply set to "0".

Storage Capacity Installed, K^{c}_{vs}, `m.storage_dict['inst-cap-c'][(min(m.stf), sit, sto, com)]`: The parameter K^{c}_{vs} represents the amount of energy content capacity of a storage s storing commodity c in a site v and support timeframe y, that is already installed to the energy system at the beginning of the model horizon. The unit of this parameter is MWh. The related section for this parameter in the spreadsheet corresponding to the first support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "inst-cap-c" represents the parameters K^{c}_{vs} for corresponding storage tuples.

Storage Capacity Upper Bound, $\overline{K}^{\mathrm{c}}_{yvs}$, `m.storage_dict['cap-up-c'][(stf, sit, sto, com)]`: The parameter $\overline{K}^{\mathrm{c}}_{yvs}$ represents the maximum amount of energy content capacity allowed of a storage s storing a commodity c in a site v in support timeframe y. The unit of this parameter is MWh. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "cap-up-c" represents the parameters $\overline{K}^{\mathrm{c}}_{yvs}$ for corresponding storage tuples. If there is no desired maximum limit for the storage energy content capacities, this parameter can be simply set to ""inf"".

Storage Power Lower Bound, $\underline{K}^{\mathrm{p}}_{yvs}$, `m.storage_dict['cap-lo-p'][(stf, sit, sto, com)]`: The parameter $\underline{K}^{\mathrm{p}}_{yvs}$ represents the minimum amount of charging/discharging power required for a storage s storing a commodity c in a site v in support timeframe y. The unit of this parameter is MW. The related section for this parameter in the spreadsheet can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "cap-lo-p" represents the parameters $\underline{K}^{\mathrm{p}}_{yvs}$ for corresponding storage tuples. If there is no desired minimum limit for the storage charging/discharging powers, this parameter can be simply set to "0".

Storage Power Installed, K^{p}_{vs}, `m.storage_dict['inst-cap-p'][(min(m.stf), sit, sto, com)]`: The parameter K^{p}_{vs} represents the amount of charging/discharging power of a storage s storing commodity c in a site v and support timeframe y, that is already installed to the energy system at the beginning of the model horizon. The unit of this parameter is MW. The related section for this parameter in the spreadsheet corresponding to the first support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "inst-cap-p" represents the parameters K^{p}_{vs} for corresponding storage tuples.

Storage Power Upper Bound, $\overline{K}^{\mathrm{p}}_{yvs}$, `m.storage_dict['cap-up-p'][(stf, sit, sto,`

com)]: The parameter $\overline{K}^{\text{c}}_{yvs}$ represents the maximum amount of charging/discharging power allowed of a storage s storing a commodity c in a site v in support timeframe y. The unit of this parameter is MW. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "cap-up-p" represents the parameters $\overline{K}^{\text{p}}_{yvs}$ for corresponding storage tuples. If there is no desired maximum limit for the storage energy content capacities, this parameter can be simply set to ""inf"".

Remaining lifetime of installed storages, T_{vs}, `m.storage.loc[(min(m.stf), sit, pro), 'lifetime']`: The parameter T_{vs} represents the remaining lifetime of already installed units. It is used to determine the set *m.inst_sto_tuples*, i.e. to identify for which support timeframes the installed units can still be used.

Storage Energy to Power Ratio, $k^{\text{E/P}}_{yvs}$, `m.storage_dict['ep-ratio'][(stf, sit, sto, com)]`: The parameter $k^{\text{E/P}}_{yvs}$ represents the linear ratio between the energy and power capacities of a storage s storing a commodity c in a site v in support timeframe y. The unit of this parameter is hours. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Storage" sheet. Here each row represents a storage technology s in a site v that stores a commodity c. The column with the header label "ep-ratio" represents the parameters $k^{\text{E/P}}_{yvs}$ for corresponding storage tuples. If there is no desired set ratio for the storage energy and power capacities (which means the storage energy and power capacities can be sized independently from each other), this cell can be left empty.

Storage New Capacity Block, $K^{\text{c,block}}_{yvs}$, `m.storage_dict['c-block'][(stf, sit, sto, com)]`: The parameter $K^{\text{c,block}}_{yvs}$ represents the capacity of all newly installed units of a storage s at a site v in the support timeframe y. The unit of this parameter is MWh. The related section for this parameter in the spreadsheet can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "c-block" represents the parameters $K^{\text{c,block}}_{yvs}$ of the corresponding storage s and site v combinations.

Storage New Power Block, $K^{\text{p,block}}_{yvs}$, `m.storage_dict['p-block'][(stf, sit, sto, com)]`: The parameter $K^{\text{p,block}}_{yvs}$ represents the power of all newly installed units of a storage s at a site v in the support timeframe y. The unit of this parameter is MW. The related section for this parameter in the spreadsheet can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "c-block" represents the parameters $K^{\text{p,block}}_{yvs}$ of the corresponding storage s and site v combinations.

Transmission Technical Parameters

Transmission Efficiency, e_{yaf}, `m.transmission_dict['eff'][(stf, sin, sout, tra, com)]`: The parameter e_{yaf} represents the energy efficiency of a transmission f that transfers a commodity c through an arc a in support timeframe y. Here an arc a defines the connection line from an origin site v_{out} to a destination site v_{in}. The ratio of the output energy amount to input energy amount, gives the energy efficiency of a transmission process. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Transmission" sheet. Here each row represents another combination of transmission f and arc a. The column with the header label "eff" represents the parameters e_{yaf} of the corresponding transmission tuples.

Transmission Capacity Lower Bound, $\underline{K}_{yaf}$, `m.transmission_dict['cap-lo'][(stf, sin, sout, tra, com)]`: The parameter $\underline{K}_{<af}$ represents the minimum power output capacity of a transmission f transferring a commodity c through an arc a, that the energy system model is required to have. Here an arc a defines the connection line from an origin site v_{out} to a destination site v_{in}. The unit of this parameter is MW. The related section for this parameter in the spreadsheet corresponding to

the support timeframe can be found under the "Transmission" sheet. Here each row represents another transmission f, arc a combination. The column with the header label "cap-lo" represents the parameters $\underline{K}_{yaf}$ of the corresponding transmission tuples.

Transmission Capacity Installed, K_{af}, `m.transmission_dict['inst-cap'][(min(m.stf), sin, sout, tra, com)]`: The parameter K_{af} represents the amount of power output capacity of a transmission f transferring a commodity c through an arc a, that is already installed to the energy system at the beginning of the modeling horizon. The unit of this parameter is MW. The related section for this parameter in the spreadsheet corresponding to the first support timeframe can be found under the "Transmission" sheet. Here each row represents another transmission f, arc a combination. The column with the header label "inst-cap" represents the parameters K_{af} of the transmission tuples.

Transmission Capacity Upper Bound, $\overline{K}_{yaf}$, `m.transmission_dict['cap-up'][(stf, sin, sout, tra, com)]`: The parameter $\overline{K}_{yaf}$ represents the maximum power output capacity of a transmission f transferring a commodity c through an arc a in support timeframe y, that the energy system model is allowed to have. Here an arc a defines the connection line from an origin site v_{out} to a destination site v_{in}. The unit of this parameter is MW. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Transmission" sheet. Here each row represents another transmission f, arc a combination. The column with the header label "cap-up" represents the parameters $\overline{K}_{yaf}$ of the corresponding transmission tuples.

Remaining lifetime of installed transmission, T_{af}, `m.transmission.loc[(min(m.stf), sitin, sitout, tra, com), 'lifetime']`: The parameter T_{af} represents the remaining lifetime of already installed units. It is used to determine the set *m.inst_tra_tuples*, i.e. to identify for which support timeframes the installed units can still be used.

Transmission New Capacity Block, K_{yaf}^{block}, `m.transmission_dict['tra-block'][(stf, sin, sout,tra, com)]`: The parameter K_{yaf}^{block} represents the capacity of all newly installed units of a transmission f transferring a commodity c through an arc a in support timeframe y.The unit of this parameter is MW. The related section for this parameter in the spreadsheet can be found under the "Transmission" sheet. Here each row represents another transmission f, arc a combination. The column with the header label "tra-block" represents the parameters K_{yaf}^{block} of the corresponding transmission tuples.

DCPF Transmission Technical Parameters

Selected transmission lines can be modelled with DC Power Flow and combined with the transport model in an energy system model. The following parameters are only required and included in the model when a transmission line should be modelled with DCPF.

Transmission Reactance, X_{yaf}, `m.transmission_dict['reactance'][(stf, sin, sout, tra, com)]`: The parameter X_{yaf} represents the reactance of a transmission f that transfers a commodity c through an arc a in support timeframe y. Here an arc a defines the connection line from an origin site v_{out} to a destination site v_{in}. Transmission reactance is used to calculate the power flow of DCPF transmission lines. This parameter is required to define a transmission line with the DCPF model and should be given in per unit system. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Transmission" sheet. Here each row represents another combination of transmission f and arc a. The column with the header label "reactance" represents the parameters X_{yaf} of the corresponding transmission tuples. If the parameter is left empty in the spreadsheet, the transmission line will be modelled with transport model as default.

Voltage Angle Difference Limit, $\overline{dl}_{yaf}$, `m.transmission_dict['difflimit'][(stf, sin, sout, tra, com)]`: The parameter $\overline{dl}_{yaf}$ represents the voltage angle difference limit of

a transmission f that transfers a commodity c through an arc a in support timeframe y. Here an arc a defines the connection line from an origin site v_{out} to a destination site v_{in}. The allowed maximum difference of voltage angles of sites v_{out} and v_{in} is limited with this parameter. This parameter is expected in degrees and a value between 0 and 91 is allowed. This parameter is required to define a transmission line with the DCPF model. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Transmission" sheet. Here each row represents another combination of transmission f and arc a. The column with the header label "difflimit" represents the parameters $\overline{dl}_{yaf}$ of the corresponding transmission tuples.

Transmission Base Voltage, $V_{yaf\text{base}}$, `m.transmission_dict['base_voltage'][(stf, sin, sout, tra, com)]`: The parameter $V_{yaf\text{base}}$ represents the base voltage of a transmission f that transfers a commodity c through an arc a in support timeframe y. Here an arc a defines the connection line from an origin site v_{out} to a destination site v_{in}. This parameter is used to calculate the power flow of DCPF transmission lines. This parameter is expected in kV and a value greater than 0 is allowed. This parameter is required to define a transmission line with the DCPF model. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "Transmission" sheet. Here each row represents another combination of transmission f and arc a. The column with the header label "base_voltage" represents the parameters $V_{yaf\text{base}}$ of the corresponding transmission tuples.

Demand Side Management Technical Parameters

DSM Efficiency, e_{yvc}, `m.dsm_dict['eff'][(stf, sit, com)]`: The parameter e_{yvc} represents the efficiency of the DSM process, i.e., the fraction of DSM upshift that is benefiting the system via the corresponding DSM downshifts of demand commodity c in site v and support timeframe y. The parameter is given as a fraction with "1" meaning a perfect recovery of the DSM upshift. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "DSM" sheet. Here each row represents another DSM potential for demand commodity c in site v. The column with the header label "eff" represents the parameters e_{yvc} of the corresponding DSM tuples.

DSM Delay Time, y_{yvc}, `m.dsm_dict['delay'][(stf, sit, com)]`: The delay time y_{yvc} restricts how long the time difference between an upshift and its corresponding downshifts may be for demand commodity c in site v and support timeframe y. The parameter is given in h. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "DSM" sheet. Here each row represents another DSM potential for demand commodity c in site v. The column with the header label "delay" represents the parameters y_{yvc} of the corresponding DSM tuples.

DSM Recovery Time, o_{yvc}, `m.dsm_dict['recov'][(stf, sit, com)]`: The recovery time o_{yvc} prevents the DSM system to continuously shift demand. During the recovery time, all upshifts of demand commodity c in site v and support timeframe y may not exceed the product of the delay time and the maximal upshift capacity. The parameter is given in h. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "DSM" sheet. Here each row represents another DSM potential for demand commodity c in site v. The column with the header label "recov" represents the parameters o_{yvc} of the corresponding DSM tuples. If no limitation via this parameter is desired it has to be set to values lower than the delay time y_{yvc}.

DSM Maximal Upshift Per Hour, $\overline{K}^{\text{up}}_{yvc}$, MW, `m.dsm_dict['cap-max-up'][(stf, sit, com)]`: The DSM upshift capacity $\overline{K}^{\text{up}}_{yvc}$ limits the total upshift per hour for a DSM potential of demand commodity c in site v and support timeframe y. The parameter is given in MW. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "DSM" sheet. Here each row represents another DSM potential for demand commodity c in site v. The column with the header label "cap-max-up" represents the parameters $\overline{K}^{\text{up}}_{yvc}$ of the corresponding DSM tuples.

DSM Maximal Downshift Per Hour, $\overline{K}^{\text{down}}_{yvc}$, MW, `m.dsm_dict['cap-max-do'][(stf, sit, com)]`: The DSM downshift capacity $\overline{K}^{\text{up}}_{yvc}$ limits the total downshift per hour for a DSM potential of demand commodity c in site v and support timeframe y. The parameter is given in MW. The related section for this parameter in the spreadsheet corresponding to the support timeframe can be found under the "DSM" sheet. Here each row represents another DSM potential for demand commodity c in site v. The column with the header label "cap-max-do" represents the parameters $\overline{K}^{\text{down}}_{yvc}$ of the corresponding DSM tuples.

Economic Parameters

Table 6: *Table: Economic Model Parameters*

Parameter	Unit	Description
j	_	Global Discount rate
D_y	_	Factor for any payment made in modeled year y
I_y	_	Factor for any investment made in modeled year y
$\overline{L}_{\text{cost}}$	€	Maximum total system costs (if CO2 is minimize
Commodity Economic Parameters		
k^{fuel}_{yvc}	€/MWh	Stock Commodity Fuel Costs
k^{env}_{yvc}	€/MWh	Environmental Commodity Costs
k^{bs}_{yvct}	€/MWh	Buy/Sell Commodity Buy/Sell Costs
k^{bs}_{yvc}	_	Multiplier for Buy/Sell Commodity Buy/Sell Cos
Process Economic Parameters		
i_{yvp}	_	Weighted Average Cost of Capital for Process
z_{yvp}	_	Process Depreciation Period
k^{inv}_{yvp}	€/MW	Process Capacity Investment Costs
k^{fix}_{yvp}	€/(MW a)	Annual Process Capacity Fixed Costs
k^{var}_{yvp}	€/MWh	Process Throughput Variable Costs
P^{start}_{yvp}	€/MW	Process Start-up Cost
Storage Economic Parameters		
i_{yvs}	_	Weighted Average Cost of Capital for Storage
z_{yvs}	_	Storage Depreciation Period
$k^{\text{p,inv}}_{yvs}$	€/MW	Storage Power Investment Costs
$k^{\text{p,fix}}_{yvs}$	€/(MW a)	Annual Storage Power Fixed Costs
$k^{\text{p,var}}_{yvs}$	€/MWh	Storage Power Variable Costs
$k^{\text{c,inv}}_{yvs}$	€/MWh	Storage Size Investment Costs
$k^{\text{c,fix}}_{yvs}$	€/(MWh a)	Annual Storage Size Fixed Costs
$k^{\text{c,var}}_{yvs}$	€/MWh	Storage Usage Variable Costs
Transmission Economic Parameters		
i_{yvf}	_	Weighted Average Cost of Capital for Transmissi
z_{yaf}	_	Tranmission Depreciation Period
k^{inv}_{yaf}	€/MW	Transmission Capacity Investment Costs
k^{fix}_{yaf}	€/(MW a)	Annual Transmission Capacity Fixed Costs
k^{var}_{yaf}	€/MWh	Tranmission Usage Variable Costs

Discount rate, j, `m.global_prop.xs('Discount rate', level=1).loc[m.global_prop.index.min()[0]]['value']`: The discount rate j is used to calculate the present value of future costs. It is set in the worksheet "Global" in the input file of the first support

timeframe.

Factor for future payments, D_y: The parameter D_y is a multiplier that has to be factored into all cost terms apart from the invest costs in intertemporal planning based on support timeframes. All other cost terms for the support timeframe y are muliplied directly with this factor to find the present value of the sum of costs in support timeframe y and all non-modeled time frames until the next modeled time frame y_{+1}, which are identical to the support timeframe with the modeling approach taken:

$$D_y = (1+j)^{1-(y-y_{\min})} \cdot \frac{1-(1+j)^{-(y_{+1}-y+1)}}{j}$$

In script `modelhelper.py` the factor D_y is implemented as the product of the functions:

```
def discount_factor(stf, m):
    """Discount for any payment made in the year stf
    """
    discount = (m.global_prop.xs('Discount rate', level=1)
                .loc[m.global_prop.index.min()[0]]['value'])

    return (1 + discount) ** (1 - (stf - m.global_prop.index.min()[0]))
```

and

```
def effective_distance(dist, m):
    """Factor for variable, fuel, purchase, sell, and fix costs.
    Calculated by repetition of modeled stfs and discount utility.
    """
    discount = (m.global_prop.xs('Discount rate', level=1)
                .loc[m.global_prop.index.min()[0]]['value'])

    if discount == 0:
        return dist
    else:
        return (1 - (1 + discount) ** (-dist)) / discount
```

Factor for investment made in support timeframe y, I_y: The parameter I_y is a multiplier that has to be factored into the invest costs in intertemporal planning based on support timeframes. The book value of the total invest costs per capacity in support timeframe y is muliplied with this factor to find the present value of the sum of costs of all annual payments made for this investment within the modeling horizon. The calculation of this parameter requires several case distinctions and is given by:

- $i \neq 0,\ j \neq 0$:

$$I_y = (1+j)^{1-(y-y_{\min})} \cdot \frac{i}{j} \cdot \left(\frac{1+i}{1+j}\right)^n \cdot \frac{(1+j)^n - (1+j)^{n-k}}{(1+i)^n - 1}$$

- $i = 0,\ j = 0$:

$$I_y = \frac{k}{n}$$

- $i \neq 0,\ j = 0$:

$$I_y = k \cdot \frac{(1+i)^n \cdot i}{(1+i)^n - 1}$$

- $i = 0,\ j \neq 0$:

$$I_y = \frac{1}{n} \cdot (1+j)^{-m} \cdot \frac{(1+j)^k - 1}{(1+j)^k \cdot j}$$

where k is the number of annualized payments that have to be made within the modeling horizon, n the depreciation period and i the weighted average cost of capital. Note that the parameters i and n take different values for different unit tuples.

In script `modelhelper.py` the factor I_y is implemented with the function:

```
def invcost_factor(dep_prd, interest, discount=None, year_built=None,
                   stf_min=None):
    """Investment cost factor formula.
    Evaluates the factor multiplied to the invest costs
    for depreciation duration and interest rate.
    Args:
        dep_prd: depreciation period (years)
        interest: interest rate (e.g. 0.06 means 6 %)
        year_built: year utility is built
        discount: discount rate for intertmeporal planning
    """
    # invcost factor for non intertemporal planning
    if discount is None:
        if interest == 0:
            return 1 / dep_prd
        else:
            return ((1 + interest) ** dep_prd * interest /
                    ((1 + interest) ** dep_prd - 1))
    # invcost factor for intertemporal planning
    elif discount == 0:
        if interest == 0:
            return 1
        else:
            return (dep_prd * ((1 + interest) ** dep_prd * interest) /
                    ((1 + interest) ** dep_prd - 1))
    else:
        if interest == 0:
            return ((1 + discount) ** (1 - (year_built-stf_min)) *
                    ((1 + discount) ** dep_prd - 1) /
                    (dep_prd * discount * (1 + discount) ** dep_prd))
        else:
            return ((1 + discount) ** (1 - (year_built-stf_min)) *
                    (interest * (1 + interest) ** dep_prd *
                    ((1 + discount) ** dep_prd - 1)) /
                    (discount * (1 + discount) ** dep_prd *
                    ((1+interest) ** dep_prd - 1)))
```

In this formulation also payments after the modeled time horizon are being made. To fix this the overpay is subtracted via:

```
def overpay_factor(dep_prd, interest, discount, year_built, stf_min, stf_
↪end):
    """Overpay value factor formula.
    Evaluates the factor multiplied to the invest costs
    for all annuity payments of a unit after the end of the
    optimization period.
    Args:
        dep_prd: depreciation period (years)
        interest: interest rate (e.g. 0.06 means 6 %)
        year_built: year utility is built
        discount: discount rate for intertemporal planning
```

(continues on next page)

(continued from previous page)

```
        k: operational time after simulation horizon
    """

    op_time = (year_built + dep_prd) - stf_end - 1

    if discount == 0:
        if interest == 0:
            return op_time / dep_prd
        else:
            return (op_time * ((1 + interest) ** dep_prd * interest) /
                    ((1 + interest) ** dep_prd - 1))
    else:
        if interest == 0:
            return ((1 + discount) ** (1 - (year_built - stf_min)) *
                    ((1 + discount) ** op_time - 1) /
                    (dep_prd * discount * (1 + discount) ** dep_prd))
        else:
            return ((1 + discount) ** (1 - (year_built - stf_min)) *
                    (interest * (1 + interest) ** dep_prd *
                    ((1 + discount) ** op_time - 1)) /
                    (discount * (1 + discount) ** dep_prd *
                    ((1 + interest) ** dep_prd - 1)))
```

In case of negative values this overpay factor is set to zero afterwards.

Maximum total system cost, $\overline{L}_{\text{cost}}$, `m.global_prop.loc[(min(m.stf), 'Cost budget'), 'value']`: This parameter restricts the total present costs over the entire modeling horizon. It is only sensible and active when the objective is a minimization of CO2 emissions.

Commodity Economic Parameters

Stock Commodity Fuel Costs, k_{vc}^{fuel}, `m.commodity_dict['price'][c]`: The parameter k_{yvc}^{fuel} represents the book cost for purchasing one unit (1 MWh) of a stock commodity c ($\forall c \in C_{\text{stock}}$) in modeled timeframe y in a site v ($\forall v \in V$). The unit of this parameter is €/MWh. The related section for this parameter in the spreadsheet belonging the support timeframe y can be found in the "Commodity" sheet. Here each row represents another commodity tuple c_{yvq} and the column of stock commodity tuples ($\forall q = "Stock"$) in this sheet with the header label "price" represents the corresponding parameter k_{yvc}^{fuel}.

Environmental Commodity Costs, k_{yvc}^{env}, `m.commodity_dict['price'][c]`: The parameter k_{yvc}^{env} represents the book cost for producing/emitting one unit (1 t, 1 kg, ...) of an environmental commodity c ($\forall c \in C_{\text{env}}$) in support timeframe y in a site v ($\forall v \in V$). The unit of this parameter is €/t (i.e. per unit of output). The related section for this parameter in the spreadsheet corresponding to support timeframe y is the "Commodity" sheet. Here, each row represents a commodity tuple c_{yvq} and the fourth column of environmental commodity tuples ($\forall q = "Env"$) in this sheet with the header label "price" represents the corresponding parameter k_{yvc}^{env}.

Buy/Sell Commodity Buy/Sell Costs, k_{yvct}^{bs}, `m.buy_sell_price_dict[c[2], ][(c[0], tm)]`: The parameter k_{yvct}^{bs} represents the purchase/buy cost for purchasing/selling one unit (1 MWh) of a buy/sell commodity c ($\forall c \in C_{\text{buy}}$)/($\forall c \in C_{\text{sell}}$) in support timeframe y in a site v ($\forall v \in V$) at timestep t ($\forall t \in T_m$). The unit of this parameter is €/MWh. The related section for this parameter in the spreadsheet can be found in the "Buy-Sell-Price" sheet. Here each column represents a commodity tuple and the row values provide the timestep information.

Multiplyer for Buy/Sell Commodity Buy/Sell Costs, k^{bs}_{yvc}, `m.commodity_dict['price'][c]`: The parameter k^{bs}_{yvc} is a multiplier for the buy/sell time series. It represents the factor on the purchase/buy cost for purchasing/selling one unit (1 MWh) of a buy/sell commodity c ($\forall c \in C_{\text{buy}}$)/($\forall c \in C_{\text{sell}}$) in support timeframe y in a site v ($\forall v \in V$). This parameter is unitless. The related section for this parameter in the spreadsheet belonging to support timeframe y can be found in the "Commodity" sheet. Here each row represents another commodity tuple c_{yvq} and the column of Buy/Sell commodity tuples ($\forall q = "Buy/Sell"$) in this sheet with the header label "price" represents the corresponding parameter k^{bs}_{yvc}.

Process Economic Parameters

Weighted Average Cost of Capital for Process, i_{yvp}, : The parameter i_{yvp} represents the weighted average cost of capital for a process technology p in support timeframe ;math:y in a site v. The weighted average cost of capital gives the interest rate (%) of costs for capital after taxes. The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Process" sheet. Here each row represents another process tuple and the column with the header label "wacc" represents the parameters i_{yvp}. The parameter is given as a percentage, where "0.07" means 7%

Process Depreciation Period, z_{yvp}: The parameter z_{yvp} represents the depreciation period of a process p built in support timeframe y in a site v. The depreciation period gives the economic and technical lifetime of a process investment. It thus features in the calculation of the invest cost factor and determines the end of operation of the process. The unit of this parameter is "a", where "a" represents a year of 8760 hours. The related section for this parameter in the spreadsheet can be found under the "Process" sheet. Here each row represents another process tuple and the column with the header label "depreciation" represents the parameters z_{yvp}.

Process Capacity Investment Costs, k^{inv}_{yvp}, `m.process_dict['inv-cost'][p]`: The parameter k^{inv}_{yvp} represents the book value of the investment cost for adding one unit new capacity of a process technology p in support timeframe y in a site v. The unit of this parameter is €/MW. To get the full impact of the investment within the modeling horizon this parameter is multiplied with the factor for the investment made in modeled year y I_y. The process capacity investment cost is to be given as an input by the user. The related section for the process capacity investment cost in the spreadsheet representing the support timeframe y can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "inv-cost" represents the process capacity investment costs of the corresponding process p and site v combinations.

Process Capacity Fixed Costs, k^{fix}_{yvp}, `m.process_dict['fix-cost'][p]`: The parameter k^{fix}_{yvp} represents the fix cost per one unit capacity κ_{yvp} of a process technology p in support timeframe y in a site v, that is charged annually. The unit of this parameter is €/(MW a). The related section for this parameter in the spreadsheet correesponding to the support timeframe y can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "fix-cost" represents the parameters k^{fix}_{yvp} of the corresponding process p and site v combinations.

Process Variable Costs, k^{var}_{yvp}, `m.process_dict['var-cost'][p]`: The parameter k^{var}_{yvp} represents the book value of the variable cost per one unit energy throughput τ_{yvpt} through a process technology p in a site v in support timeframe y. The unit of this parameter is €/MWh. The related section for this parameter in the spreadsheet corresponding to the support timeframe y can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "var-cost" represents the parameters k^{var}_{yvp} of the corresponding process p and site v combinations.

Process Start-up Cost, P^{start}_{yvp}, `m.process_dict['start-cost'][(stf, sit, pro)]`: The parameter P^{start}_{yvp} represents the cost inquired by the starting of a process p at a site v in the support timeframe y. The unit of this parameter is the currency used in the support timeframe y. The related

section for this parameter in the spreadsheet can be found under the "Process" sheet. Here each row represents another process p in a site v and the column with the header label "start-cost" represents the parameters P^{start}_{yvp} of the corresponding process p and site v combinations.

Storage Economic Parameters

Weighted Average Cost of Capital for Storage, i_{yvs}, : The parameter i_{yvs} represents the weighted average cost of capital for a storage technology s in a site v and support timeframe y. The weighted average cost of capital gives the interest rate(%) of costs for capital after taxes. The related section for this parameter in the spreadsheet corresponding to the support timeframe y can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "wacc" represents the parameters i_{yvs} of the corresponding storage s and site v combinations. The parameter is given as a percentage, where "0.07" means 7%.

Storage Depreciation Period, z_{yvs}, (a): The parameter z_{yvs} represents the depreciation period of a storage s in a site v built in support timeframe y. The depreciation period gives the economic and technical lifetime of a storage investment. It thus features in the calculation of the invest cost factor and determines the end of operation of the storage. The unit of this parameter is "a", where "a" represents a year of 8760 hours. The related section for this parameter in the spreadsheet corresponding to the support timeframe y can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "depreciation" represents the parameters z_{yvs} of the corresponding storage s and site v combinations.

Storage Power Investment Costs, $k^{\text{p,inv}}_{yvs}$, `m.storage_dict['inv-cost-p'][s]`: The parameter $k^{\text{p,inv}}_{yvs}$ represents the book value of the total investment cost for adding one unit new power output capacity of a storage technology s in a site v in support timeframe y. The unit of this parameter is €/MW. To get the full impact of the investment within the modeling horizon this parameter is multiplied with the factor for the investment made in modeled year y I_y. The related section for the storage power output capacity investment cost in the spreadsheet corresponding to the support timeframe y can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "inv-cost-p" represents the storage power output capacity investment cost of the corresponding storage s and site v combinations.

Annual Storage Power Fixed Costs, $k^{\text{p,fix}}_{yvs}$, `m.storage_dict['fix-cost-p'][s]`: The parameter $k^{\text{p,fix}}_{yvs}$ represents the fix cost per one unit power output capacity of a storage technology s in a site v and support timeframe y, that is charged annually. The unit of this parameter is €/(MW a). The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "fix-cost-p" represents the parameters $k^{\text{p,fix}}_{yvs}$ of the corresponding storage s and site v combinations.

Storage Power Variable Costs, $k^{\text{p,var}}_{yvs}$, `m.storage_dict['var-cost-p'][s]`: The parameter $k^{\text{p,var}}_{yvs}$ represents the variable cost per unit energy, that is stored in or retrieved from a storage technology s in a site v in support timeframe y. The unit of this parameter is €/MWh. The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "var-cost-p" represents the parameters $k^{\text{p,var}}_{yvs}$ of the corresponding storage s and site v combinations.

Storage Size Investment Costs, $k^{\text{c,inv}}_{yvs}$, `m.storage_dict['inv-cost-c'][s]`: The parameter $k^{\text{c,inv}}_{yvs}$ represents the book value of the total investment cost for adding one unit new storage capacity to a storage technology s in a site v in support timeframe y. The unit of this parameter is €/MWh. To get the full impact of the investment within the modeling horizon this parameter is multiplied with

the factor for the investment made in modeled year y I_y. The related section for the storage content capacity investment cost in the spreadsheet corresponding to support timeframe y can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "inv-cost-c" represents the storage content capacity investment cost of the corresponding storage s and site v combinations.

Annual Storage Size Fixed Costs, $k^{\text{c,fix}}_{yvs}$, `m.storage_dict['fix-cost-c'][s]`: The parameter $k^{\text{c,fix}}_{yvs}$ represents the fix cost per year per one unit storage content capacity of a storage technology s in a site v in support timeframe y. The unit of this parameter is €/(MWh a). The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "fix-cost-c" represents the parameters $k^{\text{c,fix}}_{vs}$ of the corresponding storage s and site v combinations.

Storage Usage Variable Costs, $k^{\text{c,var}}_{yvs}$, `m.storage_dict['var-cost-c'][s]`: The parameter $k^{\text{p,var}}_{yvs}$ represents the variable cost per unit energy, that is conserved in a storage technology s in a site v in support timeframe y. The unit of this parameter is €/MWh. The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the column with the header label "var-cost-c" represents the parameters $k^{\text{c,var}}_{yvs}$ of the corresponding storage s and site v combinations. The value of this parameter is usually set to zero, but the parameter can be taken advantage of if the storage has a short term usage or has an increased devaluation due to usage, compared to amount of energy stored.

Transmission Economic Parameters

Weighted Average Cost of Capital for Transmission, i_{yvf}, : The parameter i_{yvf} represents the weighted average cost of capital for a transmission f transferring commodities through an arc a built in support timeframe y. The weighted average cost of capital gives the interest rate(%) of costs for capital after taxes. The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the column with the header label "wacc" represents the parameters i_{yvf} of the corresponding transmission f and arc a combinations. The parameter is given as a percentage, where "0.07" means 7%.

Transmission Depreciation Period, z_{yaf}, (a): The parameter z_{yaf} represents the depreciation period of a transmission f transferring commodities through an arc a built in support timeframe y. The depreciation period of gives the economic and technical lifetime of a transmission investment. It thus features in the calculation of the invest cost factor and determines the end of operation of the transmission. The unit of this parameter is "a", where "a" represents a year of 8760 hours. The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the column with the header label "depreciation" represents the parameters z_{yaf} of the corresponding transmission f and arc a combinations.

Transmission Capacity Investment Costs, k^{inv}_{yaf}, `m.transmission_dict['inv-cost'][t]`: The parameter k^{inv}_{yaf} represents the book value of the investment cost for adding one unit new transmission capacity to a transmission f transferring commodities through an arc a in support timeframe y. To get the full impact of the investment within the modeling horizon this parameter is multiplied with the factor for the investment made in modeled year y I_y. The unit of this parameter is €/MW. The related section for the transmission capacity investment cost in the spreadsheet corresponding to support timeframe y can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the column with the header label "inv-cost" represents the transmission capacity investment cost of the corresponding transmission f and arc a combinations.

Annual Transmission Capacity Fixed Costs, k^{fix}_{yaf}, `m.transmission_dict['fix-cost'][t]`: The parameter k^{fix}_{yaf} represents the annual fix cost per one unit capacity of a transmission f transferring commodities through an arc a. The unit of this parameter is €/(MW a). The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the column with the header label "fix-cost" represents the parameters k^{fix}_{yaf} of the corresponding transmission f and arc a combinations.

Transmission Usage Variable Costs, k^{var}_{yaf}, `m.transmission_dict['var-cost'][t]`: The parameter k^{var}_{yaf} represents the variable cost per unit energy, that is transferred with a transmission f through an arc a. The unit of this parameter is €/ MWh. The related section for this parameter in the spreadsheet corresponding to support timeframe y can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the column with the header label "var-cost" represents the parameters k^{var}_{af} of the corresponding transmission f and arc a combinations.

Equations

Objective function

There are two possible choices of objective function for urbs problems, either the costs (default option) or the total CO2-emissions can be minimized.

If the total CO2-emissions are minimized the objective function takes the form:

$$w \sum_{t \in T_{\text{m}}} \sum_{v \in V} -\text{CB}(v, CO_2, t)$$

In script `model.py` the global CO2 emissions are defined and calculated by the following code fragment:

```
def co2_rule(m):
    co2_output_sum = 0
    for stf in m.stf:
        for tm in m.tm:
            for sit in m.sit:
                # minus because negative commodity_balance represents
                # creation of that commodity.
                co2_output_sum += (- commodity_balance
                                   (m, tm, stf, sit, 'CO2') *
                                   m.weight *
                                   stf_dist(stf, m))

    return (co2_output_sum)
```

In the default case the total system costs are minimized. These variable total system costs ζ are calculated by the cost function. The cost function is the objective function of the optimization model. Minimizing the value of the variable total system cost would give the most reasonable solution for the modelled energy system. The formula of the cost function expressed in mathematical notation is as following:

)The calculation of the variable total system cost is given in `model.py` by the following code fragment.

```
def cost_rule(m):
    if m.type =='sub':
        return (pyomo.summation(m.costs) + sum(0.5 * m.rho[(tm, stf, sit_
↪in, sit_out)] *
                        (m.e_tra_in[(tm, stf, sit_in,sit_out, tra, com)]
                        -m.flow_global[(tm, stf, sit_in,sit_out)])**2
                        for tm in m.tm
                        for stf, sit_in, sit_out, tra, com in m.tra_tuples_
↪boun) + sum(m.lamda[(tm, stf, sit_in, sit_out)] *
                        (m.e_tra_in[(tm,stf, sit_in,sit_out,tra,com)]
                        -m.flow_global[(tm, stf, sit_in,sit_out)])
                        for tm in m.tm
                        for stf, sit_in, sit_out, tra, com in m.tra_tuples_
↪boun)      )
    else:
        return pyomo.summation(m.costs)
```

The variable total system cost ζ is basically calculated by the summation of every type of total costs. As previously mentioned in section *Cost Types*, these cost types are : `Investment`, `Fix`, `Variable`, `Fuel`, `Revenue`, `Purchase`, `Start-up` and `Environmental`.

In script `model.py` the individual cost functions are calculated by the following code fragment:

```
def def_costs_rule(m, cost_type):
    #Calculate total costs by cost type.
    #Sums up process activity and capacity expansions
    #and sums them in the cost types that are specified in the set
    #m.cost_type. To change or add cost types, add/change entries
    #there and modify the if/elif cases in this function accordingly.
    #Cost types are
    #  - Investment costs for process power, storage power and
    #    storage capacity. They are multiplied by the investment
    #    factors. Rest values of units are subtracted.
    #  - Fixed costs for process power, storage power and storage
    #    capacity.
    #  - Variables costs for usage of processes, storage and transmission.
    #  - Fuel costs for stock commodity purchase.

    if cost_type == 'Invest':
        cost = \
            sum(m.cap_pro_new[p] *
                m.process_dict['inv-cost'][p] *
                m.process_dict['invcost-factor'][p]
                for p in m.pro_tuples)
        if m.mode['int']:
            cost += \
                sum(m.cap_pro_new[p] *
                    m.process_dict['inv-cost'][p] *
                    m.process_dict['overpay-factor'][p]
                    for p in m.pro_tuples)
        if m.mode['tra']:
            # transmission_cost is defined in transmission.py
            cost += transmission_cost(m, cost_type)
        if m.mode['sto']:
            # storage_cost is defined in storage.py
            cost += storage_cost(m, cost_type)
```

(continues on next page)

(continued from previous page)

```
        return m.costs[cost_type] == cost

    elif cost_type == 'Fixed':
        cost = \
            sum(m.cap_pro[p] * m.process_dict['fix-cost'][p] *
                m.process_dict['cost_factor'][p]
                for p in m.pro_tuples)
        if m.mode['tra']:
            cost += transmission_cost(m, cost_type)
        if m.mode['sto']:
            cost += storage_cost(m, cost_type)
        return m.costs[cost_type] == cost

    elif cost_type == 'Variable':
        cost = \
            sum(m.tau_pro[(tm,) + p] * m.weight *
                m.process_dict['var-cost'][p] *
                m.process_dict['cost_factor'][p]
                for tm in m.tm
                for p in m.pro_tuples)
        if m.mode['tra']:
            cost += transmission_cost(m, cost_type)
        if m.mode['sto']:
            cost += storage_cost(m, cost_type)
        return m.costs[cost_type] == cost

    elif cost_type == 'Fuel':
        return m.costs[cost_type] == sum(
            m.e_co_stock[(tm,) + c] * m.weight *
            m.commodity_dict['price'][c] *
            m.commodity_dict['cost_factor'][c]
            for tm in m.tm for c in m.com_tuples
            if c[2] in m.com_stock)

    elif cost_type == 'Start-up':
        if m.mode['onoff']:
            cost = sum(m.start_up[(tm,) + p] * m.weight *
                   m.start_price_dict[p] * m.cap_pro[p] *
                   m.process_dict['cost_factor'][p]
                   for tm in m.tm
                   for p in m.pro_start_up_tuples)
            return m.costs[cost_type] == cost
        else:
            return m.costs[cost_type] == 0



    elif cost_type == 'Environmental':
        return m.costs[cost_type] == sum(
            - commodity_balance(m, tm, stf, sit, com) * m.weight *
            m.commodity_dict['price'][(stf, sit, com, com_type)] *
            m.commodity_dict['cost_factor'][(stf, sit, com, com_type)]
            for tm in m.tm
            for stf, sit, com, com_type in m.com_tuples
            if com in m.com_env)
```

(continues on next page)

(continued from previous page)

```
    # Revenue and Purchase costs defined in BuySellPrice.py
    elif cost_type == 'Revenue':
        return m.costs[cost_type] == revenue_costs(m)

    elif cost_type == 'Purchase':
        return m.costs[cost_type] == purchase_costs(m)

    else:
        raise NotImplementedError("Unknown cost type.")
```

Constraints

Commodity Constraints

Commodity Balance The function commodity balance calculates the in- and outflows into all processes, storages and transmission of a commodity c in a site v in support timeframe y at a timestep t. The value of the function CB being greater than zero $\mathrm{CB} > 0$ means that the presence of the commodity c in the site v in support timeframe y at the timestep t is getting by the interaction with the technologies given above. Correspondingly, the value of the function being less than zero means that the presence of the commodity in the site at the timestep is getting more than before by the technologies given above. The mathematical explanation of this rule for general problems is explained in *Energy Storage*.

In script `modelhelper.py` the value of the commodity balance function $\mathrm{CB}(y, v, c, t)$ is calculated by the following code fragment:

```
def commodity_balance(m, tm, stf, sit, com):
    """Calculate commodity balance at given timestep.
    For a given commodity co and timestep tm, calculate the balance of
    consumed (to process/storage/transmission, counts positive) and␣
↪provided
    (from process/storage/transmission, counts negative) commodity flow.␣
↪Used
    as helper function in create_model for constraints on demand and stock
    commodities.
    Args:
        m: the model object
        tm: the timestep
        site: the site
        com: the commodity
    Returns
        balance: net value of consumed (positive) or provided (negative)␣
↪power
    """
    balance = (sum(m.e_pro_in[(tm, stframe, site, process, com)]
                   # usage as input for process increases balance
                   for stframe, site, process in m.pro_tuples
                   if site == sit and stframe == stf and
                   (stframe, process, com) in m.r_in_dict) -
               sum(m.e_pro_out[(tm, stframe, site, process, com)]
                   # output from processes decreases balance
                   for stframe, site, process in m.pro_tuples
                   if site == sit and stframe == stf and
                   (stframe, process, com) in m.r_out_dict))
```

(continues on next page)

(continued from previous page)

```
    if m.mode['tra']:
        balance += transmission_balance(m, tm, stf, sit, com)
    if m.mode['sto']:
        balance += storage_balance(m, tm, stf, sit, com)

    return balance
```

where the two functions introducing the partly balances for transmissions and storages, respectively, are given by:

```
def transmission_balance(m, tm, stf, sit, com):
    """called in commodity balance
    For a given commodity co and timestep tm, calculate the balance of
    import and export """

    return (sum(m.e_tra_in[(tm, stframe, site_in, site_out,
                            transmission, com)]
                # exports increase balance
                for stframe, site_in, site_out, transmission, commodity
                in m.tra_tuples
                if (site_in == sit and stframe == stf and
                    commodity == com)) -
            sum(m.e_tra_out[(tm, stframe, site_in, site_out,
                             transmission, com)]
                # imports decrease balance
                for stframe, site_in, site_out, transmission, commodity
                in m.tra_tuples
                if (site_out == sit and stframe == stf and
                    commodity == com)))
```

```
def storage_balance(m, tm, stf, sit, com):
    """callesd in commodity balance
    For a given commodity co and timestep tm, calculate the balance of
    storage input and output """

    return sum(m.e_sto_in[(tm, stframe, site, storage, com)] -
               m.e_sto_out[(tm, stframe, site, storage, com)]
               # usage as input for storage increases consumption
               # output from storage decreases consumption
               for stframe, site, storage, commodity in m.sto_tuples
               if site == sit and stframe == stf and commodity == com)
```

Vertex Rule: The vertex rule is the main constraint that has to be satisfied for every commodity. It represents a version of "Kirchhoff's current law" or local energy conservation. This constraint is defined differently for each commodity type. The inequality requires, that any imbalance (CB>0, CB<0) of a commodity c in a site v and support timeframe y at a timestep t to be balanced by a corresponding source term or demand. The rule is not defined for environmental or SupIm commodities. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint vertex rule is defined and calculated by the following code fragments:

```
m.res_vertex = pyomo.Constraint(
    m.tm, m.com_tuples,
    rule=res_vertex_rule,
    doc='storage + transmission + process + source + buy - sell == demand')
```

```
def res_vertex_rule(m, tm, stf, sit, com, com_type):
    # environmental or supim commodities don't have this constraint (yet)
    if com in m.com_env:
        return pyomo.Constraint.Skip
    if com in m.com_supim:
        return pyomo.Constraint.Skip

    # helper function commodity_balance calculates balance from input to
    # and output from processes, storage and transmission.
    # if power_surplus > 0: production/storage/imports create net positive
    #                       amount of commodity com
    # if power_surplus < 0: production/storage/exports consume a net
    #                       amount of the commodity com
    power_surplus = - commodity_balance(m, tm, stf, sit, com)

    # if com is a stock commodity, the commodity source term e_co_stock
    # can supply a possibly negative power_surplus
    if com in m.com_stock:
        power_surplus += m.e_co_stock[tm, stf, sit, com, com_type]

    # if Buy and sell prices are enabled
    if m.mode['bsp']:
        power_surplus += bsp_surplus(m, tm, stf, sit, com, com_type)

    # if com is a demand commodity, the power_surplus is reduced by the
    # demand value; no scaling by m.dt or m.weight is needed here, as this
    # constraint is about power (MW), not energy (MWh)
    if com in m.com_demand:
        try:
            power_surplus -= m.demand_dict[(sit, com)][(stf, tm)]
        except KeyError:
            pass

    if m.mode['dsm']:
        power_surplus += dsm_surplus(m, tm, stf, sit, com)

    return power_surplus == 0
```

where the two functions introducing the effects of Buy/Sell or DSM events, respectively, are given by:

```
def bsp_surplus(m, tm, stf, sit, com, com_type):

    power_surplus = 0

    # if com is a sell commodity, the commodity source term e_co_sell
    # can supply a possibly positive power_surplus
    if com in m.com_sell:
        power_surplus -= m.e_co_sell[tm, stf, sit, com, com_type]

    # if com is a buy commodity, the commodity source term e_co_buy
    # can supply a possibly negative power_surplus
    if com in m.com_buy:
        power_surplus += m.e_co_buy[tm, stf, sit, com, com_type]

    return power_surplus
```

```
def dsm_surplus(m, tm, stf, sit, com):
    """ called in vertex rule
        calculate dsm surplus"""
    if (stf, sit, com) in m.dsm_site_tuples:
        return (- m.dsm_up[tm, stf, sit, com] +
                sum(m.dsm_down[t, tm, stf, sit, com]
                    for t in dsm_time_tuples(
                    tm, m.timesteps[1:],
                    max(int(1 / m.dt *
                        m.dsm_dict['delay'][(stf, sit, com)]), 1))))
    else:
        return 0
```

Stock Per Step Rule: The constraint stock per step rule applies only for commodities of type "Stock" ($c \in C_{\mathrm{st}}$). This constraint limits the amount of stock commodity $c \in C_{\mathrm{st}}$, that can be used by the energy system in the site v in support timeframe y at the timestep t. This amount is limited by the product of the parameter maximum stock supply limit per hour $\bar{l}_{yvc}$ and the timestep length Δt. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint stock per step rule is defined and calculated by the following code fragment:

```
m.res_stock_step = pyomo.Constraint(
    m.tm, m.com_tuples,
    rule=res_stock_step_rule,
    doc='stock commodity input per step <= commodity.maxperstep')
```

```
def res_stock_step_rule(m, tm, stf, sit, com, com_type):
    if com not in m.com_stock:
        return pyomo.Constraint.Skip
    else:
        return (m.e_co_stock[tm, stf, sit, com, com_type] <=
                m.dt * m.commodity_dict['maxperhour']
                [(stf, sit, com, com_type)])
```

Total Stock Rule: The constraint total stock rule applies only for commodities of type "Stock" ($c \in C_{\mathrm{st}}$). This constraint limits the amount of stock commodity $c \in C_{\mathrm{st}}$, that can be used annually by the energy system in the site v and support timeframe y. This amount is limited by the parameter maximum annual stock supply limit per vertex $\overline{L}_{yvc}$. The annual usage of stock commodity is calculated by the sum of the products of the parameter weight w and the parameter stock commodity source term ρ_{yvct}, summed over all timesteps $t \in T_m$. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint total stock rule is defined and calculated by the following code fragment:

```
m.res_stock_total = pyomo.Constraint(
    m.com_tuples,
    rule=res_stock_total_rule,
    doc='total stock commodity input <= commodity.max')
```

```
def res_stock_total_rule(m, stf, sit, com, com_type):
    if com not in m.com_stock:
        return pyomo.Constraint.Skip
    else:
```

(continues on next page)

(continued from previous page)

```
        # calculate total consumption of commodity com
        total_consumption = 0
        for tm in m.tm:
            total_consumption += (
                m.e_co_stock[tm, stf, sit, com, com_type])
        total_consumption *= m.weight
        return (total_consumption <=
                m.commodity_dict['max'][(stf, sit, com, com_type)])
```

Sell Per Step Rule: The constraint sell per step rule applies only for commodities of type "Sell" ($c \in C_{\text{sell}}$). This constraint limits the amount of sell commodity $c \in C_{\text{sell}}$, that can be sold by the energy system in the site v in support timeframe y at the timestep t. The limit is defined by the parameter maximum sell supply limit per hour $\overline{g}_{yvc}$. To satisfy this constraint, the value of the variable sell commodity source term ϱ_{yvct} must be less than or equal to the value of the parameter maximum sell supply limit per hour $\overline{g}_{vc}$ multiplied with the length of the time steps Δt. The mathematical explanation of this rule is given in *Trading with an external market*.

In script `BuySellPrice.py` the constraint sell per step rule is defined and calculated by the following code fragment:

```
m.res_sell_step = pyomo.Constraint(
   m.tm, m.com_tuples,
   rule=res_sell_step_rule,
   doc='sell commodity output per step <= commodity.maxperstep')
```

```
def res_sell_step_rule(m, tm, stf, sit, com, com_type):
    if com not in m.com_sell:
        return pyomo.Constraint.Skip
    else:
        return (m.e_co_sell[tm, stf, sit, com, com_type] <=
                m.dt * m.commodity_dict['maxperhour']
                [(stf, sit, com, com_type)])
```

Total Sell Rule: The constraint total sell rule applies only for commodities of type "Sell" ($c \in C_{\text{sell}}$). This constraint limits the amount of sell commodity $c \in C_{\text{sell}}$, that can be sold annually by the energy system in the site v and support timeframe y. The limit is defined by the parameter maximum annual sell supply limit per vertex $\overline{G}_{yvc}$. The annual usage of sell commodity is calculated by the sum of the products of the parameter weight w and the parameter sell commodity source term ϱ_{yvct}, summed over all timesteps $t \in T_m$. The mathematical explanation of this rule is given in *Trading with an external market*.

In script `BuySellPrice.py` the constraint total sell rule is defined and calculated by the following code fragment:

```
m.res_sell_total = pyomo.Constraint(
    m.com_tuples,
    rule=res_sell_total_rule,
    doc='total sell commodity output <= commodity.max')
```

```
def res_sell_total_rule(m, stf, sit, com, com_type):
    if com not in m.com_sell:
        return pyomo.Constraint.Skip
    else:
        # calculate total sale of commodity com
```

(continues on next page)

(continued from previous page)

```
        total_consumption = 0
        for tm in m.tm:
            total_consumption += (
                m.e_co_sell[tm, stf, sit, com, com_type])
        total_consumption *= m.weight
        return (total_consumption <=
                m.commodity_dict['max'][(stf, sit, com, com_type)])
```

Buy Per Step Rule: The constraint buy per step rule applies only for commodities of type "Buy" ($c \in C_{\text{buy}}$). This constraint limits the amount of buy commodity $c \in C_{\text{buy}}$, that can be bought by the energy system in the site v in support timeframe y at the timestep t. The limit is defined by the parameter maximum buy supply limit per time step $\overline{b}_{yvc}$. To satisfy this constraint, the value of the variable buy commodity source term ψ_{yvct} must be less than or equal to the value of the parameter maximum buy supply limit per time step $\overline{b}_{vc}$, multiplied by the length of the time steps Δt. The mathematical explanation of this rule is given in *Trading with an external market*.

In script `BuySellPrice.py` the constraint buy per step rule is defined and calculated by the following code fragment:

```
m.res_buy_step = pyomo.Constraint(
    m.tm, m.com_tuples,
    rule=res_buy_step_rule,
    doc='buy commodity output per step <= commodity.maxperstep')
```

```
def res_buy_step_rule(m, tm, stf, sit, com, com_type):
    if com not in m.com_buy:
        return pyomo.Constraint.Skip
    else:
        return (m.e_co_buy[tm, stf, sit, com, com_type] <=
                m.dt * m.commodity_dict['maxperhour']
                [(stf, sit, com, com_type)])
```

Total Buy Rule: The constraint total buy rule applies only for commodities of type "Buy" ($c \in C_{\text{buy}}$). This constraint limits the amount of buy commodity $c \in C_{\text{buy}}$, that can be bought annually by the energy system in the site v in support timeframe y. The limit is defined by the parameter maximum annual buy supply limit per vertex $\overline{B}_{yvc}$. To satisfy this constraint, the annual usage of buy commodity must be less than or equal to the value of the parameter buy supply limit per vertex $\overline{B}_{vc}$. The annual usage of buy commodity is calculated by the sum of the products of the parameter weight w and the parameter buy commodity source term ψ_{yvct}, summed over all modeled timesteps $t \in T_m$. The mathematical explanation of this rule is given in *Trading with an external market*.

In script `BuySellPrice.py` the constraint total buy rule is defined and calculated by the following code fragment:

```
m.res_buy_total = pyomo.Constraint(
   m.com_tuples,
   rule=res_buy_total_rule,
   doc='total buy commodity output <= commodity.max')
```

```
def res_buy_total_rule(m, stf, sit, com, com_type):
    if com not in m.com_buy:
        return pyomo.Constraint.Skip
    else:
```

(continues on next page)

(continued from previous page)

```
        # calculate total sale of commodity com
        total_consumption = 0
        for tm in m.tm:
            total_consumption += (
                m.e_co_buy[tm, stf, sit, com, com_type])
        total_consumption *= m.weight
        return (total_consumption <=
                m.commodity_dict['max'][(stf, sit, com, com_type)])
```

Environmental Output Per Step Rule: The constraint environmental output per step rule applies only for commodities of type "Env" ($c \in C_{\text{env}}$). This constraint limits the amount of environmental commodity $c \in C_{\text{env}}$, that can be released to environment by the energy system in the site v in support timeframe y at the timestep t. The limit is defined by the parameter maximum environmental output per time step $\overline{m}_{yvc}$. To satisfy this constraint, the negative value of the commodity balance for the given environmental commodity $c \in C_{\text{env}}$ must be less than or equal to the value of the parameter maximum environmental output per time step $\overline{m}_{vc}$, multiplied by the length of the time steps Δt. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint environmental output per step rule is defined and calculated by the following code fragment:

```
m.res_env_step = pyomo.Constraint(
    m.tm, m.com_tuples,
    rule=res_env_step_rule,
    doc='environmental output per step <= commodity.maxperstep')
```

```
def res_env_step_rule(m, tm, stf, sit, com, com_type):
    if com not in m.com_env:
        return pyomo.Constraint.Skip
    else:
        environmental_output = - commodity_balance(m, tm, stf, sit, com)
        return (environmental_output <=
                m.dt * m.commodity_dict['maxperhour']
                [(stf, sit, com, com_type)])
```

Total Environmental Output Rule: The constraint total environmental output rule applies only for commodities of type "Env" ($c \in C_{\text{env}}$). This constraint limits the amount of environmental commodity $c \in C_{\text{env}}$, that can be released to environment annually by the energy system in the site v in support timeframe y. The limit is defined by the parameter maximum annual environmental output limit per vertex $\overline{M}_{yvc}$. To satisfy this constraint, the annual release of environmental commodity must be less than or equal to the value of the parameter maximum annual environmental output $\overline{M}_{vc}$. The annual release of environmental commodity is calculated by the sum of the products of the parameter weight w and the negative value of commodity balance function, summed over all modeled time steps $t \in T_m$. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint total environmental output rule is defined and calculated by the following code fragment:

```
m.res_env_total = pyomo.Constraint(
    m.com_tuples,
    rule=res_env_total_rule,
    doc='total environmental commodity output <= commodity.max')
```

```
def res_env_total_rule(m, stf, sit, com, com_type):
    if com not in m.com_env:
        return pyomo.Constraint.Skip
    else:
        # calculate total creation of environmental commodity com
        env_output_sum = 0
        for tm in m.tm:
            env_output_sum += (- commodity_balance(m, tm, stf, sit, com))
        env_output_sum *= m.weight
        return (env_output_sum <=
                m.commodity_dict['max'][(stf, sit, com, com_type)])
```

Demand Side Management Constraints

The DSM equations are taken from the Paper of Zerrahn and Schill "On the representation of demand-side management in power system models", DOI: 10.1016/j.energy.2015.03.037.

DSM Variables Rule: The DSM variables rule defines the relation between the up- and downshifted DSM commodities. An upshift $\delta^{\text{up}}_{yvct}$ in site v and support timeframe y of demand commodity c in time step t can be compensated during a certain time step interval $[t - y_{yvc}/\Delta t, t + y_{yvc}/\Delta t]$ by multiple downshifts $\delta^{\text{down}}_{t,tt,yvc}$. Here, y_{yvc} represents the allowable delay time of downshifts in hours, which is scaled into time steps by dividing by the timestep length Δt. Depending on the DSM efficiency e_{yvc}, an upshift in a DSM commodity may correspond to multiple downshifts which sum to less than the original upshift. The mathematical explanation of this rule is given in *Demand side management*.

In script `dsm.py` the constraint DSM variables rule is defined by the following code fragment:

```
m.def_dsm_variables = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=def_dsm_variables_rule,
    doc='DSMup * efficiency factor n == DSMdo (summed)')
```

```
def def_dsm_variables_rule(m, tm, stf, sit, com):
    dsm_down_sum = 0
    for tt in dsm_time_tuples(tm,
                              m.timesteps[1:],
                              max(int(1 / m.dt *
                                  m.dsm_dict['delay'][(stf, sit, com)]),␣
↪1)):
        dsm_down_sum += m.dsm_down[tm, tt, stf, sit, com]
    return dsm_down_sum == (m.dsm_up[tm, stf, sit, com] *
                            m.dsm_dict['eff'][(stf, sit, com)])
```

DSM Upward Rule: The DSM upshift $\delta^{\text{up}}_{yvct}$ in site v and support timeframe y of demand commodity c in time step t is limited by the DSM maximal upshift per hour $\overline{K}^{\text{up}}_{yvc}$, multiplied by the length of the time steps Δt. The mathematical explanation of this rule is given in *Demand side management*.

In script `dsm.py` the constraint DSM upward rule is defined by the following code fragment:

```
m.res_dsm_upward = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=res_dsm_upward_rule,
    doc='DSMup <= Cup (threshold capacity of DSMup)')
```

```
def res_dsm_upward_rule(m, tm, stf, sit, com):
    return m.dsm_up[tm, stf, sit, com] <= (m.dt *
                                           m.dsm_dict['cap-max-up']
                                           [(stf, sit, com)])
```

DSM Downward Rule: The total DSM downshift $\delta^{\text{down}}_{t,tt,yvc}$ in site v and support timeframe y of demand commodity c in time step t is limited by the DSM maximal downshift per hour $\overline{K}^{\text{down}}_{yvc}$, multiplied by the length of the time steps Δt. The mathematical explanation of this rule is given in *Demand side management*.

In script `dsm.py` the constraint DSM downward rule is defined by the following code fragment:

```
m.res_dsm_downward = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=res_dsm_downward_rule,
    doc='DSMdo (summed) <= Cdo (threshold capacity of DSMdo)')
```

```
def res_dsm_downward_rule(m, tm, stf, sit, com):
    dsm_down_sum = 0
    for t in dsm_time_tuples(tm,
                             m.timesteps[1:],
                             max(int(1 / m.dt *
                                 m.dsm_dict['delay'][(stf, sit, com)]),␣
↪1)):
        dsm_down_sum += m.dsm_down[t, tm, stf, sit, com]
    return dsm_down_sum <= (m.dt * m.dsm_dict['cap-max-do'][(stf, sit,␣
↪com)])
```

DSM Maximum Rule: The DSM maximum rule limits the shift of one DSM unit in site v in support timeframe y of demand commodity c in time step t. The mathematical explanation of this rule is given in *Demand side management*.

In script `dsm.py` the constraint DSM maximum rule is defined by the following code fragment:

```
m.res_dsm_maximum = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=res_dsm_maximum_rule,
    doc='DSMup + DSMdo (summed) <= max(Cup,Cdo)')
```

```
def res_dsm_maximum_rule(m, tm, stf, sit, com):
    dsm_down_sum = 0
    for t in dsm_time_tuples(tm,
                             m.timesteps[1:],
                             max(int(1 / m.dt *
                                 m.dsm_dict['delay'][(stf, sit, com)]),␣
↪1)):
        dsm_down_sum += m.dsm_down[t, tm, stf, sit, com]

    max_dsm_limit = m.dt * max(m.dsm_dict['cap-max-up'][(stf, sit, com)],
                               m.dsm_dict['cap-max-do'][(stf, sit, com)])
    return m.dsm_up[tm, stf, sit, com] + dsm_down_sum <= max_dsm_limit
```

DSM Recovery Rule: The DSM recovery rule limits the upshift in site v and support timeframe y of demand commodity c during a set recovery period o_{yvc}. Since the recovery period o_{yvc} is input as hours, it is scaled into time steps by dividing it by the length of the time steps Δt. The mathematical explanation of this rule is given in *Demand side management*.

In script `dsm.py` the constraint DSM Recovery rule is defined by the following code fragment:

```
m.res_dsm_recovery = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=res_dsm_recovery_rule,
    doc='DSMup(t, t + recovery time R) <= Cup * delay time L')
```

```
def res_dsm_recovery_rule(m, tm, stf, sit, com):
    dsm_up_sum = 0
    for t in dsm_recovery(tm,
                          m.timesteps[1:],
                          max(int(1 / m.dt *
                              m.dsm_dict['recov'][(stf, sit, com)]), 1)):
        dsm_up_sum += m.dsm_up[t, stf, sit, com]
    return dsm_up_sum <= (m.dsm_dict['cap-max-up'][(stf, sit, com)] *
                          m.dsm_dict['delay'][(stf, sit, com)])
```

Global Environmental Constraint

Global CO2 Limit Rule: The constraint global CO2 limit rule applies to the whole energy system in one support timeframe y, that is to say it applies to every site and timestep. This constraints restricts the total amount of CO2 to environment. The constraint states that the sum of released CO2 across all sites $v \in V$ and timesteps $t \in t_m$ must be less than or equal to the parameter maximum global annual CO2 emission limit $\overline{L}_{CO_2,y}$, where the amount of released CO2 in a single site v at a single timestep t is calculated by the product of commodity balance of environmental commodities $\mathrm{CB}(y, v, CO_2, t)$ and the parameter weight w. This constraint is skipped if the value of the parameter $\overline{L}_{CO_2}$ is set to `inf`. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint annual global CO2 limit rule is defined and calculated by the following code fragment:

```
def res_global_co2_limit_rule(m, stf):
    if math.isinf(m.global_prop_dict['value'][stf, 'CO2 limit']):
        return pyomo.Constraint.Skip
    elif m.global_prop_dict['value'][stf, 'CO2 limit'] >= 0:
        co2_output_sum = 0
        for tm in m.tm:
            for sit in m.sit:
                # minus because negative commodity_balance represents␣
↪creation
                # of that commodity.
                co2_output_sum += (- commodity_balance(m, tm,
                                                       stf, sit, 'CO2'))

        # scaling to annual output (cf. definition of m.weight)
        co2_output_sum *= m.weight
        return (co2_output_sum <= m.global_prop_dict['value']
                                                    [stf, 'CO2 limit'])
    else:
        return pyomo.Constraint.Skip
```

Global CO2 Budget Rule: The constraint global CO2 budget rule applies to the whole energy system over the entire modeling horizon, that is to say it applies to every support timeframe, site and timestep. This constraints restricts the total amount of CO2 to environment. The constraint states that the sum of

released CO2 across all support timeframe $y \in Y$, sites $v \in V$ and timesteps $t \in t_m$ must be less than or equal to the parameter maximum global CO2 emission budget $\overline{\overline{L}}_{CO_2,y}$, where the amount of released CO2 in a single support timeframe y in a single site v and at a single timestep t is calculated by the product of the commodity balance of environmental commodities $\mathrm{CB}(y, v, CO_2, t)$ and the parameter weight w. This constraint is skipped if the value of the parameter $\overline{\overline{L}}_{CO_2}$ is set to `inf`. The mathematical explanation of this rule is given in *Intertemporal optimization model*.

In script `model.py` the constraint global CO2 budget is defined and calculated by the following code fragment:

```
def res_global_co2_budget_rule(m):
    if math.isinf(m.global_prop_dict['value'][min(m.stf_list), 'CO2 budget
↪']):
        return pyomo.Constraint.Skip
    elif (m.global_prop_dict['value'][min(m.stf_list), 'CO2 budget']) >= 0:
        co2_output_sum = 0
        for stf in m.stf:
            for tm in m.tm:
                for sit in m.sit:
                    # minus because negative commodity_balance represents
                    # creation of that commodity.
                    co2_output_sum += (- commodity_balance
                                       (m, tm, stf, sit, 'CO2') *
                                       m.weight *
                                       stf_dist(stf, m))

        return (co2_output_sum <=
                m.global_prop_dict['value'][min(m.stf), 'CO2 budget'])
    else:
        return pyomo.Constraint.Skip
```

Process Constraints

Process Capacity Rule: The constraint process capacity rule defines the variable total process capacity κ_{yvp}. The variable total process capacity is defined by the constraint as the sum of the parameter process capacity installed K_{vp} and the variable new process capacity $\hat{\kappa}_{yvp}$. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint process capacity rule is defined and calculated by the following code fragment:

```
m.def_process_capacity = pyomo.Constraint(
    m.pro_tuples,
    rule=def_process_capacity_rule,
    doc='total process capacity = inst-cap + new capacity')
```

```
def def_process_capacity_rule(m, stf, sit, pro):
    if m.mode['int']:
        if (sit, pro, stf) in m.inst_pro_tuples:
            if (sit, pro, min(m.stf)) in m.pro_const_cap_dict:
                cap_pro = m.process_dict['inst-cap'][(stf, sit, pro)]
            else:
                cap_pro = \
                    (sum(m.cap_pro_new[stf_built, sit, pro]
```

(continues on next page)

(continued from previous page)

```
                        for stf_built in m.stf
                        if (sit, pro, stf_built, stf)
                        in m.operational_pro_tuples) +
                    m.process_dict['inst-cap'][(min(m.stf), sit, pro)])
        else:
            cap_pro = sum(
                m.cap_pro_new[stf_built, sit, pro]
                for stf_built in m.stf
                if (sit, pro, stf_built, stf) in m.operational_pro_tuples)
    else:
        if (sit, pro, stf) in m.pro_const_cap_dict:
            cap_pro = m.process_dict['inst-cap'][(stf, sit, pro)]
        else:
            cap_pro = (m.cap_pro_new[stf, sit, pro] +
                       m.process_dict['inst-cap'][(stf, sit, pro)])
    return cap_pro
```

Process Input Rule: The constraint process input rule defines the variable process input commodity flow $\epsilon^{\text{in}}_{yvcpt}$. The variable process input commodity flow is defined by the constraint as the product of the variable process throughput τ_{yvpt} and the parameter process input ratio r^{in}_{ypc}.The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint process input rule is defined and calculated by the following code fragment:

```
m.def_process_input = pyomo.Constraint(
    m.tm, m.pro_input_tuples - m.pro_partial_input_tuples,
    rule=def_process_input_rule,
    doc='process input = process throughput * input ratio')
```

```
def def_process_input_rule(m, tm, stf, sit, pro, com):
    return (m.e_pro_in[tm, stf, sit, pro, com] ==
            m.tau_pro[tm, stf, sit, pro] * m.r_in_dict[(stf, pro, com)])
```

Process Output Rule: The constraint process output rule defines the variable process output commodity flow $\epsilon^{\text{out}}_{yvcpt}$. The variable process output commodity flow is defined by the constraint as the product of the variable process throughput τ_{yvpt} and the parameter process output ratio r^{out}_{ypc}. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint process output rule is defined and calculated by the following code fragment:

```
m.def_process_output = pyomo.Constraint(
    m.tm, (m.pro_output_tuples - m.pro_partial_output_tuples -
           m.pro_timevar_output_tuples),
    rule=def_process_output_rule,
    doc='process output = process throughput * output ratio')
```

```
def def_process_output_rule(m, tm, stf, sit, pro, com):
    return (m.e_pro_out[tm, stf, sit, pro, com] ==
            m.tau_pro[tm, stf, sit, pro] * m.r_out_dict[(stf, pro, com)])
```

Intermittent Supply Rule: The constraint intermittent supply rule defines the variable process input commodity flow $\epsilon^{\text{in}}_{yvcpt}$ for processes p that use a supply intermittent commodity $c \in C_{\text{sup}}$ as input. Therefore this constraint only applies if a commodity is an intermittent supply commodity $c \in C_{\text{sup}}$.

The variable process input commodity flow is defined by the constraint as the product of the variable total process capacity κ_{yvp} and the parameter intermittent supply capacity factor s_{yvct}, scaled by the size of the time steps :math: *Delta t*. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint intermittent supply rule is defined and calculated by the following code fragment:

```
m.def_intermittent_supply = pyomo.Constraint(
    m.tm, m.pro_input_tuples,
    rule=def_intermittent_supply_rule,
    doc='process output = process capacity * supim timeseries')
```

```
def def_intermittent_supply_rule(m, tm, stf, sit, pro, coin):
    if coin in m.com_supim:
        return (m.e_pro_in[tm, stf, sit, pro, coin] ==
                m.cap_pro[stf, sit, pro] * m.supim_dict[(sit, coin)]
                [(stf, tm)] * m.dt)
    else:
        return pyomo.Constraint.Skip
```

Process Throughput By Capacity Rule: The constraint process throughput by capacity rule limits the variable process throughput τ_{yvpt}. This constraint prevents processes from exceeding their capacity. The constraint states that the variable process throughput must be less than or equal to the variable total process capacity κ_{yvp}, scaled by the size of the time steps :math: *Delta t*. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint process throughput by capacity rule is defined and calculated by the following code fragment:

```
m.res_process_throughput_by_capacity = pyomo.Constraint(
    m.tm, m.pro_tuples,
    rule=res_process_throughput_by_capacity_rule,
    doc='process throughput <= total process capacity')
```

```
def res_process_throughput_by_capacity_rule(m, tm, stf, sit, pro):
    return (m.tau_pro[tm, stf, sit, pro] <= m.dt * m.cap_pro[stf, sit,␣
↪pro])
```

Process Throughput Gradient Rule: The constraint process throughput gradient rule limits the process power gradient $\left|\tau_{yvpt} - \tau_{yvp(t-1)}\right|$. This constraint prevents processes from exceeding their maximal possible change in activity from one time step to the next. The constraint states that the absolute power gradient must be less than or equal to the maximal power ramp up gradient $overline PG^{\text{up}}_{yvp}$ parameter when increasing power or to the maximal power ramp down gradient $\overline{PG}^{\text{up}}_{yvp}$ parameter (both scaled to capacity and by time step duration). The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint process throughput gradient rule is split into 2 parts and defined and calculated by the following code fragments:

```
m.res_process_rampdown = pyomo.Constraint(
    m.tm, m.pro_rampdowngrad_tuples,
    rule=res_process_rampdown_rule,
    doc='throughput may not decrease faster than maximal ramp down gradient
↪')
```

(continues on next page)

(continued from previous page)

```
m.res_process_rampup = pyomo.Constraint(
    m.tm, m.pro_rampupgrad_tuples,
    rule=res_process_rampup_rule,
    doc='throughput may not increase faster than maximal ramp up gradient')
```

```
def res_process_rampdown_rule(m, t, stf, sit, pro):
    return (m.tau_pro[t - 1, stf, sit, pro] -
            m.cap_pro[stf, sit, pro] *
            m.process_dict['ramp-down-grad'][(stf, sit, pro)] * m.dt <=
            m.tau_pro[t, stf, sit, pro])
```

```
def res_process_rampup_rule(m, t, stf, sit, pro):
    return (m.tau_pro[t - 1, stf, sit, pro] +
            m.cap_pro[stf, sit, pro] *
            m.process_dict['ramp-up-grad'][(stf, sit, pro)] * m.dt >=
            m.tau_pro[t, stf, sit, pro])
```

Process Capacity Limit Rule: The constraint process capacity limit rule limits the variable total process capacity κ_{yvp}. This constraint restricts a process p in a site v and support timeframe y from having more total capacity than an upper bound and having less than a lower bound. The constraint states that the variable total process capacity κ_{yvp} must be greater than or equal to the parameter process capacity lower bound $\underline{K}_{yvp}$ and less than or equal to the parameter process capacity upper bound $\overline{K}_{yvp}$. The mathematical explanation of this rule is given in *Minimal optimization model*.

In script `model.py` the constraint process capacity limit rule is defined and calculated by the following code fragment:

```
m.res_process_capacity = pyomo.Constraint(
    m.pro_tuples,
    rule=res_process_capacity_rule,
    doc='process.cap-lo <= total process capacity <= process.cap-up')
```

```
def res_process_capacity_rule(m, stf, sit, pro):
    return (m.process_dict['cap-lo'][stf, sit, pro],
            m.cap_pro[stf, sit, pro],
            m.process_dict['cap-up'][stf, sit, pro])
```

Sell Buy Symmetry Rule: The constraint sell buy symmetry rule defines the total process capacity κ_{yvp} of a process p in a site v and support timeframe y that uses either sell or buy commodities ($c \in C_{\text{sell}} \vee C_{\text{buy}}$), therefore this constraint only applies to processes that use sell or buy commodities. The constraint states that the total process capacities κ_{yvp} of processes that use complementary buy and sell commodities must be equal. Buy and sell commodities are complementary, when a commodity c is an output of a process where the buy commodity is an input, and at the same time the commodity c is an input commodity of a process where the sell commodity is an output. The mathematical explanation of this rule is given in *Trading with an external market*.

In script `BuySellPrice.py` the constraint sell buy symmetry rule is defined and calculated by the following code fragment:

```
m.res_sell_buy_symmetry = pyomo.Constraint(
    m.pro_input_tuples,
    rule=res_sell_buy_symmetry_rule,
    doc='total power connection capacity must be symmetric in both '
        'directions')
```

```
def res_sell_buy_symmetry_rule(m, stf, sit_in, pro_in, coin):
    # constraint only for sell and buy processes
    # and the processes must be in the same site
    if coin in m.com_buy:
        sell_pro = search_sell_buy_tuple(m, stf, sit_in, pro_in, coin)
        if sell_pro is None:
            return pyomo.Constraint.Skip
        else:
            return (m.cap_pro[stf, sit_in, pro_in] ==
                    m.cap_pro[stf, sit_in, sell_pro])
    else:
        return pyomo.Constraint.Skip
```

Process time variable output rule: This constraint multiplies the process efficiency with the parameter time series f^{out}_{yvpt}. The process output for all commodities is thus manipulated depending on time. This constraint is not valid for environmental commodities since these are typically linked to an input commodity flow rather than an output commodity flow. The mathematical explanation of this rule is given in *Advanced Processes*.

In script `AdvancedProcesses.py` the constraint process time variable output rule is defined and calculated by the following code fragment:

```
m.def_process_timevar_output = pyomo.Constraint(
    m.tm, m.pro_timevar_output_tuples,
    rule=def_pro_timevar_output_rule,
    doc='e_pro_out = tau_pro * r_out * eff_factor')
```

```
def def_pro_timevar_output_rule(m, tm, stf, sit, pro, com):
    return (m.e_pro_out[tm, stf, sit, pro, com] ==
            m.tau_pro[tm, stf, sit, pro] * m.r_out_dict[(stf, pro, com)] *
            m.eff_factor_dict[(sit, pro)][stf, tm])
```

Process Constraints for partial operation

The process constraints for partial operation described in the following are only activated if in the input file there is a value set in the column **ratio-min** for an **input commodity** or for an **output commodity** in the **process-commodity** sheet for the process in question.

It is important to understand that this partial load formulation can only work if its accompanied by a non-zero value for the minimum partial load fraction $\underline{P}_{yvp}$.

Without activating the on/off feature in the **process** sheet, the partial load feature can only be used for processes that are never meant to be shut down and are always operating only between a given part load state and full load. Please see the next chapter for the combined on/off and partial operation features.

Throughput by Min fraction Rule: This constraint limits the minimal operational state of a process downward, making sure that the minimal part load fraction is honored. The mathematical explanation of this rule is given in *Advanced Processes*.

In script `AdvancedProcesses.py` this constraint is defined and calculated by the following code fragment:

```
m.res_throughput_by_capacity_min = pyomo.Constraint(
    m.tm, m.pro_partial_tuples,
```

(continues on next page)

(continued from previous page)

```
    rule=res_throughput_by_capacity_min_rule,
    doc='cap_pro * min-fraction <= tau_pro')
```

```
def res_throughput_by_capacity_min_rule(m, tm, stf, sit, pro):
    return (m.tau_pro[tm, stf, sit, pro] >=
            m.cap_pro[stf, sit, pro] *
            m.process_dict['min-fraction'][(stf, sit, pro)] * m.dt)
```

Partial Process Input Rule: The link between operational state tau_{yvpt} and commodity in/outputs is changed from a simple linear behavior to a more complex one. Instead of constant in- and output ratios these are now interpolated linearly between the value for full operation $r_{yvp}^{\text{in/out}}$ at full load and the minimum in/output ratios $\underline{r}_{yvp}^{\text{in/out}}$ at the minimum operation point. The mathematical explanation of this rule is given in *Advanced Processes*.

In script *model.py* this expression is written in the following way for the input ratio (and analogous for the output ratios):

```
m.def_partial_process_input = pyomo.Constraint(
    m.tm, m.pro_partial_input_tuples,
    rule=def_partial_process_input_rule,
    doc='e_pro_in = cap_pro * min_fraction * (r - R) / (1 - min_fraction)'
                '+ tau_pro * (R - min_fraction * r) / (1 - min_fraction)')
```

```
def def_partial_process_input_rule(m, tm, stf, sit, pro, com):
    # input ratio at maximum operation point
    R = m.r_in_dict[(stf, pro, com)]
    # input ratio at lowest operation point
    r = m.r_in_min_fraction_dict[stf, pro, com]
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]

    online_factor = min_fraction * (r - R) / (1 - min_fraction)
    throughput_factor = (R - min_fraction * r) / (1 - min_fraction)
    return (m.e_pro_in[tm, stf, sit, pro, com] ==
            m.dt * m.cap_pro[stf, sit, pro] * online_factor +
            m.tau_pro[tm, stf, sit, pro] * throughput_factor)
```

In case of a process where also a time variable output efficiency is given the code for the output changes to.

```
m.def_process_partial_timevar_output = pyomo.Constraint(
    m.tm, m.pro_partial_output_tuples & m.pro_timevar_output_tuples,
    rule=def_pro_partial_timevar_output_rule,
    doc='e_pro_out = tau_pro * r_out * eff_factor')
```

```
def def_pro_partial_timevar_output_rule(m, tm, stf, sit, pro, com):
# input ratio at maximum operation point
    R = m.r_out_dict[stf, pro, com]
    # input ratio at lowest operation point
    r = m.r_out_min_fraction_dict[stf, pro, com]
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]

    online_factor = min_fraction * (r - R) / (1 - min_fraction)
    throughput_factor = (R - min_fraction * r) / (1 - min_fraction)
    return (m.e_pro_out[tm, stf, sit, pro, com] ==
```

(continues on next page)

(continued from previous page)

```
            (m.dt * m.cap_pro[stf, sit, pro] * online_factor +
             m.tau_pro[tm, stf, sit, pro] * throughput_factor) *
             m.eff_factor_dict[(sit, pro)][stf, tm])
```

Process Constraints for the on/off feature

The process constraints for the on/off feature described in this chapter are only activated if, in the input file, the value „1" is set is set in the column **on-off** for a process in the **process** sheet.

Process Throughput and On/Off Coupling Rule: These two constraints couple the variables process throughput τ_{yvpt} and process on/off marker $_{yvpt}$. This is done by turning the marker on (boolean value 1) when the throughput is greater than the minimum load of the process.The mathematical explanation of this rule is given in *Advanced Processes*.

In script `AdvancedProcesses.py` this constraint is defined and calculated by the following code fragment:

```
m.res_throughput_by_on_off_lower = pyomo.Constraint(
    m.tm, m.pro_on_off_tuples | m.pro_partial_on_off_tuples,
    rule=res_throughput_by_on_off_lower_rule,
    doc='tau_pro >= min-fraction * cap_pro * on_off')
m.res_throughput_by_on_off_upper = pyomo.Constraint(
    m.tm, m.pro_on_off_tuples | m.pro_partial_on_off_tuples,
    rule=res_throughput_by_on_off_upper_rule,
    doc='tau_pro <='
        'cap_pro * on_off + min-fraction * cap_pro * (1 - on_off)')
```

```
def res_throughput_by_on_off_lower_rule(m, tm, stf, sit, pro):
    return (m.tau_pro[tm, stf, sit, pro] >=
            m.min_fraction_dict[stf, sit, pro] * m.cap_pro[stf, sit, pro] *
            m.dt * m.on_off[tm, stf, sit, pro])
```

```
def res_throughput_by_on_off_upper_rule(m, tm, stf, sit, pro):
    return (m.tau_pro[tm, stf, sit, pro] <=
            m.cap_pro[stf, sit, pro] * m.dt * m.on_off[tm, stf, sit, pro] +
            m.min_fraction_dict[stf, sit, pro] * m.cap_pro[stf, sit, pro] *
            m.dt * (1 - m.on_off[tm, stf, sit, pro]))
```

Process On/Off Output Rule: This constraint modifies the process output commodity flow $\epsilon^{\text{out}}_{yvcpt}$ when compared to the original version without the on/off feature in two ways by differentiating between the output **commodity type** q. When the **commodity type** is `Env`, the output remains the same as without the on/off feature. Otherwise, the original output equation is multiplied with the variable process on/off marker $_{yvpt}$. The mathematical explanation of this rule is given in *Advanced Processes*.

In script `AdvancedProcesses.py` the constraint process on/off output rule is defined and calculated by the following code fragment:

```
m.def_process_on_off_output = pyomo.Constraint(
    m.tm, m.pro_on_off_output_tuples - m.pro_timevar_output_tuples -
          m.pro_partial_on_off_output_tuples,
    rule=def_process_on_off_output_rule,
    doc='e_pro_out = tau_pro * r_out * on_off')
```

```
def def_process_on_off_output_rule(m, tm, stf, sit, pro, com):
    r = m.r_out_dict[(stf, pro, com)]
    if com in m.com_env:
        return (m.e_pro_out[tm, stf, sit, pro, com] ==
                m.tau_pro[tm, stf, sit, pro] * r)
    else:
        return (m.e_pro_out[tm, stf, sit, pro, com] ==
                m.tau_pro[tm, stf, sit, pro] * r * m.on_off[tm, stf, sit,␣
↪pro])
```

In the case of a process where also a time variable output efficiency is given the code for the output changes to:

```
m.def_process_on_off_timevar_output = pyomo.Constraint(
    m.tm, m.pro_timevar_output_tuples & m.pro_on_off_output_tuples -
          m.pro_partial_on_off_output_tuples,
    rule=def_process_on_off_timevar_output_rule,
    doc='e_pro_out == tau_pro * r_out * on_off * eff_factor')
```

```
def def_process_on_off_timevar_output_rule(m, tm, stf, sit, pro, com):
    return (m.e_pro_out[tm, stf, sit, pro, com] ==
            m.tau_pro[tm, stf, sit, pro] * m.r_out_dict[(stf, pro, com)] *
            m.on_off[tm, stf, sit, pro] *
            m.eff_factor_dict[(sit, pro)][stf, tm])
```

Process On/Off Partial Input Rule: This constraint modifies the process input commodity flow $\epsilon^{\text{in}}_{yvcpt}$ when compared to the original partial operation version without the on/off feature in by differentiating between two possible input functions, depending on the process on/off marker $_{yvpt}$. When the marker is on, the input function is the same as in the case of simple partial operation. When the marker is off, the input function becomes the product of the variable process throughput τ_{yvpt} and the parameter process partial input ratio $\underline{r}^{\text{in}}_{ypc}$. the output **commodity type** q. When the **commodity type**. The mathematical explanation of this rule is given in *Advanced Processes*.

In script `AdvancedProcesses.py` the constraint process on/off output rule is defined and calculated by the following code fragment:

```
m.def_partial_process_on_off_input = pyomo.Constraint(
    m.tm, m.pro_partial_on_off_input_tuples,
    rule=def_partial_process_on_off_input_rule,
    doc='e_pro_in = '
        ' (cap_pro * min_fraction * (r - R) / (1 - min_fraction)'
        ' + tau_pro * (R - min_fraction * r) / (1 - min_fraction))')
```

```
def def_partial_process_on_off_input_rule(m, tm, stf, sit, pro, com):
    # input ratio at maximum operation point
    R = m.r_in_dict[(stf, pro, com)]
    # input ratio at lowest operation point
    r = m.r_in_min_fraction_dict[stf, pro, com]
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]

    online_factor = min_fraction * (r - R) / (1 - min_fraction)
    throughput_factor = (R - min_fraction * r) / (1 - min_fraction)
    return (m.e_pro_in[tm, stf, sit, pro, com] ==
            (m.dt * m.cap_pro[stf, sit, pro] * online_factor +
             m.tau_pro[tm, stf, sit, pro] * throughput_factor) *
```

(continues on next page)

(continued from previous page)

```
                m.on_off[tm, stf, sit, pro] +
                m.tau_pro[tm, stf, sit, pro] * r *
                (1 - m.on_off[tm, stf, sit, pro]))
```

Process On/Off Partial Output Rule: This constraint modifies the process output commodity flow $\epsilon^{\text{out}}_{yvcpt}$ when compared to the original partial operation version without the on/off feature in two ways by differentiating between the output **commodity type** q. When the **commodity type** is not `Env`, the output remains the same as for the partial operation without the on/off feature. Otherwise, the original output equation is changes depending on the variable process on/off marker $_{yvpt}$. When the marker is off, the output function becomes the product of the variable process throughput τ_{yvpt} and the parameter process partial output ratio $\underline{r}^{\text{out}}_{ypc}$. When the marker is on, the output function for `Env` type commodities remains the same as for the partial operation without the on/off feature. The mathematical explanation of this rule is given in *Advanced Processes*.

```
m.def_partial_process_on_off_output = pyomo.Constraint(
    m.tm, m.pro_partial_on_off_output_tuples - m.pro_timevar_output_tuples,
    rule=def_partial_process_on_off_output_rule,
    doc='e_pro_out = on_off *'
        ' (cap_pro * min_fraction * (r - R) / (1 - min_fraction) '
        '+ tau_pro * (R - min_fraction * r) / (1 - min_fraction)) ')
```

```
def def_partial_process_on_off_output_rule(m, tm, stf, sit, pro, com):
    # input ratio at maximum operation point
    R = m.r_out_dict[stf, pro, com]
    # input ratio at lowest operation point
    r = m.r_out_min_fraction_dict[stf, pro, com]
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]
    on_off = m.on_off[tm, stf, sit, pro]

    online_factor = min_fraction * (r - R) / (1 - min_fraction)
    throughput_factor = (R - min_fraction * r) / (1 - min_fraction)
    if com in m.com_env:
        return(m.e_pro_out[tm, stf, sit, pro, com] ==
               (m.dt * m.cap_pro[stf, sit, pro] * online_factor +
               m.tau_pro[tm, stf, sit, pro] * throughput_factor) * on_off +
               m.tau_pro[tm, stf, sit, pro] * r *
               (1 - on_off))
    else:
        return (m.e_pro_out[tm, stf, sit, pro, com] ==
                (m.dt * m.cap_pro[stf, sit, pro] * online_factor +
                m.tau_pro[tm, stf, sit, pro] * throughput_factor) * on_off)
```

In the case of a process where also a time variable output efficiency is given the code for the output changes to:

```
m.def_process_partial_on_off_timevar_output = pyomo.Constraint(
    m.tm, m.pro_partial_on_off_output_tuples & m.pro_timevar_output_tuples,
    rule=def_pro_partial_on_off_timevar_output_rule,
    doc='e_pro_out == tau_pro * r_out * on_off * eff_factor')
```

```
def def_partial_process_on_off_output_rule(m, tm, stf, sit, pro, com):
    # input ratio at maximum operation point
    R = m.r_out_dict[stf, pro, com]
    # input ratio at lowest operation point
```

(continues on next page)

(continued from previous page)

```
    r = m.r_out_min_fraction_dict[stf, pro, com]
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]
    on_off = m.on_off[tm, stf, sit, pro]

    online_factor = min_fraction * (r - R) / (1 - min_fraction)
    throughput_factor = (R - min_fraction * r) / (1 - min_fraction)
    if com in m.com_env:
        return(m.e_pro_out[tm, stf, sit, pro, com] ==
               (m.dt * m.cap_pro[stf, sit, pro] * online_factor +
               m.tau_pro[tm, stf, sit, pro] * throughput_factor) * on_off +
               m.tau_pro[tm, stf, sit, pro] * r *
               (1 - on_off))
    else:
        return (m.e_pro_out[tm, stf, sit, pro, com] ==
                (m.dt * m.cap_pro[stf, sit, pro] * online_factor +
                m.tau_pro[tm, stf, sit, pro] * throughput_factor) * on_off)
```

Process Starting Ramp-up Rule: This constraint replaces the process throughput ramping rule when the parameter process starting time $\overline{ST}^{\text{start}}_{yvp}$ is defined in the input **process** sheet. This is done only until the variable process throughput τ_{yvpt} reaches the minimum load value and only while increasing the process throughput τ_{yvpt}. The mathematical explanation of this rule is given in *Advanced Processes*.

In script `AdvancedProcesses.py` the constraint process starting ramp-up rule is defined and calculated by the following code fragment:

```
m.res_starting_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_start_tuples,
    rule=res_starting_rampup_rule,
    doc='throughput may not increase faster than maximal starting ramp up '
        'gradient until reaching minimum capacity')
```

```
def res_starting_rampup_rule(m, t, stf, sit, pro):
    min_fraction = m.min_fraction_dict[stf, sit, pro]
    start_time = m.process_dict['start-time'][(stf, sit, pro)]
    starting_ramp =min_fraction / start_time
    return (m.tau_pro[t - 1, stf, sit, pro] +
            m.cap_pro[stf, sit, pro] *
            m.process_dict['ramp-up-grad'][(stf, sit, pro)] * m.dt *
            m.on_off[t - 1, stf, sit, pro] +
            m.cap_pro[stf, sit, pro] *
            starting_ramp * m.dt *
            (1 - m.on_off[t - 1, stf, sit, pro])
            >=
            m.tau_pro[t, stf, sit, pro])
```

Process Output Ramping Rule: These constraints act as a limiter for the process output $\epsilon^{\text{out}}_{yvcpt}$ with the on/off feature because the process on/off marker $_{yvpt}$ can be both on and off in the minimum load point. There are three possible cases, as follows, defined in the script `AdvanceProcesses.py`. The mathematical explanation of this rule is given in *Advanced Processes*

Case I: The parameter process minimum load fraction $\underline{P}_{yvp}$ is greater than the parameter process maximum power ramp up gradient $\overline{PG}^{\text{up}}_{yvp}$ and is divisible with it. It is defined and calculated by the following code fragment:

```
m.res_output_minfraction_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_divides_minfraction_output_tuples -
          m.pro_partial_on_off_output_tuples - m.pro_timevar_output_tuples,
    rule=res_output_minfraction_rampup_rule,
    doc='Output may not increase faster than the minimal working capacity')
```

```
def res_output_minfraction_rampup_rule(m, tm, stf, sit, pro, com):
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                m.process_dict['min-fraction'][(stf, sit, pro)] *
                m.r_out_dict[(stf, pro, com)] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has partial operation, the code changes to:

```
m.res_partial_output_minfraction_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_divides_minfraction_output_tuples &
          m.pro_partial_on_off_output_tuples - m.pro_timevar_output_tuples,
    rule=res_partial_output_minfraction_rampup_rule,
    doc='Output may not increase faster than the minimal working capacity')
```

```
def res_partial_output_minfraction_rampup_rule(m, tm, stf, sit, pro, com):
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                m.process_dict['min-fraction'][(stf, sit, pro)] *
                m.r_out_min_fraction_dict[(stf, pro, com)] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has time variable efficiency, the code changes to:

```
m.res_timevar_output_minfraction_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_divides_minfraction_output_tuples &
          m.pro_timevar_output_tuples - m.pro_partial_on_off_output_tuples,
    rule=res_timevar_output_minfraction_rampup_rule,
    doc='Output may not increase faster than the minimal working capacity')
```

```
def res_timevar_output_minfraction_rampup_rule(m, tm, stf, sit, pro, com):
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                m.process_dict['min-fraction'][(stf, sit, pro)] *
                m.r_out_dict[(stf, pro, com)] *
                m.eff_factor_dict[(sit, pro)][stf, tm] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has both partial operation and time variable efficiency, the code changes to:

```
m.res_partial_timevar_output_minfraction_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_divides_minfraction_output_tuples &
          m.pro_partial_on_off_output_tuples & m.pro_timevar_output_tuples,
    rule=res_partial_timevar_output_minfraction_rampup_rule,
    doc='Output may not increase faster than the minimal working capacity')
```

```
def res_partial_timevar_output_minfraction_rampup_rule(m, tm, stf, sit,␣
↪pro, com):
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                m.process_dict['min-fraction'][(stf, sit, pro)] *
                m.r_out_min_fraction_dict[(stf, pro, com)] *
                m.eff_factor_dict[(sit, pro)][stf, tm] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

Case II: The parameter process minimum load fraction $\underline{P}_{yvp}$ is greater than the parameter process maximum power ramp up gradient $\overline{PG}^{\text{up}}_{yvp}$, but is not divisible with it. It is defined and calculated by the following code fragment:

```
m.res_output_minfraction_rampup_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_not_divides_minfraction_output_tuples -
          m.pro_partial_on_off_output_tuples - m.pro_timevar_output_tuples,
    rule=res_output_minfraction_rampup_rampup_rule,
    doc='Output may not increase faster than the first multiple of the'
        'ramping up gradient greater than the minimal working capacity')
```

```
def res_output_minfraction_rampup_rampup_rule(m, tm, stf, sit, pro, com):
    ramp_up = m.process_dict['ramp-up-grad'][(stf, sit, pro)]
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]

    first_output_value = (math.floor(min_fraction / ramp_up) + 1) * ramp_up
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                first_output_value *
                m.r_out_dict[(stf, pro, com)] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has partial operation, the code changes to:

```
m.res_partial_output_minfraction_rampup_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_not_divides_minfraction_output_tuples &
          m.pro_partial_on_off_output_tuples - m.pro_timevar_output_tuples,
    rule=res_partial_output_minfraction_rampup_rampup_rule,
    doc='Output may not increase faster than the first multiple of the'
        'ramping up gradient greater than the minimal working capacity')
```

```
def res_partial_output_minfraction_rampup_rampup_rule(m, tm, stf, sit, pro,
↪ com):
    ramp_up = m.process_dict['ramp-up-grad'][(stf, sit, pro)]
```

(continues on next page)

(continued from previous page)

```
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]

    first_output_value = (math.floor(min_fraction / ramp_up) + 1) * ramp_up
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                first_output_value *
                m.r_out_min_fraction_dict[(stf, pro, com)] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has time variable efficiency, the code changes to:

```
m.res_timevar_output_minfraction_rampup_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_not_divides_minfraction_output_tuples &
          m.pro_timevar_output_tuples - m.pro_partial_on_off_output_tuples,
    rule=res_timevar_output_minfraction_rampup_rampup_rule,
    doc='Output may not increase faster than the first multiple of the'
        'ramping up gradient greater than the minimal working capacity')
```

```
def res_timevar_output_minfraction_rampup_rampup_rule(m, tm, stf, sit, pro,
↪ com):
    ramp_up = m.process_dict['ramp-up-grad'][(stf, sit, pro)]
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]

    first_output_value = (math.floor(min_fraction / ramp_up) + 1) * ramp_up
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                first_output_value *
                m.r_out_dict[(stf, pro, com)] *
                m.eff_factor_dict[(sit, pro)][stf, tm] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has both partial operation and time variable efficiency, the code changes to:

```
m.res_partial_timevar_output_minfraction_rampup_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_not_divides_minfraction_output_tuples &
          m.pro_partial_on_off_output_tuples & m.pro_timevar_output_tuples,
    rule=res_partial_timevar_output_minfraction_rampup_rampup_rule,
    doc='Output may not increase faster than the first multiple of the'
        'ramping up gradient greater than the minimal working capacity')
```

```
def res_partial_timevar_output_minfraction_rampup_rampup_rule(m, tm, stf,␣
↪sit, pro, com):
    ramp_up = m.process_dict['ramp-up-grad'][(stf, sit, pro)]
    min_fraction = m.process_dict['min-fraction'][(stf, sit, pro)]

    first_output_value = (math.floor(min_fraction / ramp_up) + 1) * ramp_up
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
```

(continues on next page)

(continued from previous page)

```
                first_output_value *
                m.r_out_min_fraction_dict[(stf, pro, com)] *
                m.eff_factor_dict[(sit, pro)][stf, tm] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

Case III: The parameter process minimum load fraction $\underline{P}_{yvp}$ is smaller than the parameter process maximum power ramp up gradient $\overline{PG}^{\text{up}}_{yvp}$. It is defined and calculated by the following code fragment:

```
m.res_output_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_bigger_minfraction_output_tuples -
          m.pro_partial_on_off_output_tuples - m.pro_timevar_output_tuples,
    rule=res_output_rampup_rule,
    doc='Output may not increase faster than the ramping up gradient')
```

```
def res_output_rampup_rule(m, tm, stf, sit, pro, com):
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                m.process_dict['ramp-up-grad'][(stf, sit, pro)] *
                m.r_out_dict[(stf, pro, com)] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has partial operation, the code changes to:

```
m.res_partial_output_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_bigger_minfraction_output_tuples &
          m.pro_partial_on_off_output_tuples - m.pro_timevar_output_tuples,
    rule=res_partial_output_rampup_rule,
    doc='Output may not increase faster than the ramping up gradient')
```

```
def res_partial_output_rampup_rule(m, tm, stf, sit, pro, com):
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                m.process_dict['ramp-up-grad'][(stf, sit, pro)] *
                m.r_out_min_fraction_dict[(stf, pro, com)] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has time variable efficiency, the code changes to:

```
m.res_timevar_output_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_bigger_minfraction_output_tuples &
          m.pro_timevar_output_tuples - m.pro_partial_on_off_output_tuples,
    rule=res_timevar_output_rampup_rule,
    doc='Output may not increase faster than the ramping up gradient')
```

```
def res_timevar_output_rampup_rule(m, tm, stf, sit, pro, com):
    if tm != m.timesteps[1]:
```

(continues on next page)

(continued from previous page)

```
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                m.process_dict['ramp-up-grad'][(stf, sit, pro)] *
                m.r_out_dict[(stf, pro, com)] *
                m.eff_factor_dict[(sit, pro)][stf, tm] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

If the process has both partial operation and time variable efficiency, the code changes to:

```
m.res_partial_timevar_output_rampup = pyomo.Constraint(
    m.tm, m.pro_rampup_bigger_minfraction_output_tuples &
          m.pro_partial_on_off_output_tuples & m.pro_timevar_output_tuples,
    rule=res_partial_timevar_output_rampup_rule,
    doc='Output may not increase faster than the ramping up gradient')
```

```
def res_partial_timevar_output_rampup_rule(m, tm, stf, sit, pro, com):
    if tm != m.timesteps[1]:
        return (m.e_pro_out[tm - 1, stf, sit, pro, com] +
                m.cap_pro[stf, sit, pro] * m.dt *
                m.process_dict['ramp-up-grad'][(stf, sit, pro)] *
                m.r_out_min_fraction_dict[(stf, pro, com)] *
                m.eff_factor_dict[(sit, pro)][stf, tm] >=
                m.e_pro_out[tm, stf, sit, pro, com])
    else:
        return pyomo.Constraint.Skip
```

Process Start-Up Rule: The constraint process start-up rule marks in the variable process start marker σ_{yvpt} whether a process p started in timestep t or not. The mathematical explanation of this rule is given in *Advanced Processes*.

In script `AdvancedProcesses.py` the constraint process start ups rule is defined and calculated by the following code fragment:

```
m.res_start_up = pyomo.Constraint(
    m.tm, m.pro_start_up_tuples,
    rule=res_start_ups_rule,
    doc='start >= on_off(t) - on_off(t-1)')
```

```
def res_start_up_rule(m, t, stf, sit, pro):
    return (m.start_up[t, stf, sit, pro] >= m.on_off[t, stf, sit, pro] -
                                            m.on_off[t - 1, stf, sit,␣
↪pro])
```

Transmission Constraints

Transmission Capacity Rule: The constraint transmission capacity rule defines the variable total transmission capacity κ_{yaf}. The variable total transmission capacity is defined by the constraint as the sum of the variable transmission capacity installed K_{yaf} and the variable new transmission capacity $\hat{\kappa}_{yaf}$. The mathematical explanation of this rule is given in *Multinode optimization model*.

In script `transmission.py` the constraint transmission capacity rule is defined and calculated by the

following code fragment:

```
m.def_transmission_capacity = pyomo.Constraint(
    m.tra_tuples,
    rule=def_transmission_capacity_rule,
    doc='total transmission capacity = inst-cap + new capacity')
```

```
def def_transmission_capacity_rule(m, stf, sin, sout, tra, com):
    if m.mode['int']:
        if (sin, sout, tra, com, stf) in m.inst_tra_tuples:
            if (min(m.stf), sin, sout, tra, com) in m.tra_const_cap_dict:
                cap_tra = m.transmission_dict['inst-cap'][
                    (min(m.stf), sin, sout, tra, com)]
            else:
                cap_tra = (
                    sum(m.cap_tra_new[stf_built, sin, sout, tra, com]
                        for stf_built in m.stf
                        if (sin, sout, tra, com, stf_built, stf) in
                        m.operational_tra_tuples) +
                    m.transmission_dict['inst-cap']
                    [(min(m.stf), sin, sout, tra, com)])
        else:
            cap_tra = (
                sum(m.cap_tra_new[stf_built, sin, sout, tra, com]
                    for stf_built in m.stf
                    if (sin, sout, tra, com, stf_built, stf) in
                    m.operational_tra_tuples))
    else:
        if (stf, sin, sout, tra, com) in m.tra_const_cap_dict:
            cap_tra = \
                m.transmission_dict['inst-cap'][(stf, sin, sout, tra, com)]
        else:
            cap_tra = (m.cap_tra_new[stf, sin, sout, tra, com] +
                       m.transmission_dict['inst-cap'][
                           (stf, sin, sout, tra, com)])

    return cap_tra
```

Transmission New Capacity Rule: The constraint transmission new capacity rule defines the variable new trasmission capacity $\hat{\kappa}_{yaf}$. This variable is defined by the constraint as the product of the parameter transmission new capacity block K_{yaf}^{block} and the variable new transmission capacity units β_{yaf}. The mathematical explanation of this rule is given in *Multinode optimization model*.

In script `transmission.py` the constraint transmission output rule is defined and calculated by the following code fragment:

```
m.def_cap_tra_new = pyomo.Constraint(
     m.tra_block_tuples,
     rule=def_cap_tra_new_rule,
     doc='cap_tra_new = tra-block * cap_tra_new')
```

```
def def_cap_tra_new_rule(m, stf, sin, sout, tra, com):
    return(m.cap_tra_new[stf, sin, sout, tra, com] ==
           m.tra_cap_unit[stf, sin, sout, tra, com] *
           m.transmission_dict['tra-block'][(stf, sin, sout, tra, com)])
```

Transmission Output Rule: The constraint transmission output rule defines the variable transmission

output commodity flow π^{out}_{yaft}. The variable transmission output commodity flow is defined by the constraint as the product of the variable transmission input commodity flow π^{in}_{yaft} and the parameter transmission efficiency e_{yaf}. The mathematical explanation of this rule is given in *Multinode optimization model*.

In script `transmission.py` the constraint transmission output rule is defined and calculated by the following code fragment:

```
m.def_transmission_output = pyomo.Constraint(
    m.tm, m.tra_tuples,
    rule=def_transmission_output_rule,
    doc='transmission output = transmission input * efficiency')
```

```
def def_transmission_output_rule(m, tm, stf, sin, sout, tra, com):
    return (m.e_tra_out[tm, stf, sin, sout, tra, com] ==
            m.e_tra_in[tm, stf, sin, sout, tra, com] *
            m.transmission_dict['eff'][(stf, sin, sout, tra, com)])
```

Transmission Input By Capacity Rule: The constraint transmission input by capacity rule limits the variable transmission input commodity flow π^{in}_{yaft}. This constraint prevents the transmission power from exceeding the possible power input capacity of the line. The constraint states that the variable transmission input commodity flow π^{in}_{yaft} must be less than or equal to the variable total transmission capacity κ_{yaf}, scaled by the size of the time steps :math: *Delta t*. The mathematical explanation of this rule is given in *Multinode optimization model*.

In script `transmission.py` the constraint transmission input by capacity rule is defined and calculated by the following code fragment:

```
m.res_transmission_input_by_capacity = pyomo.Constraint(
    m.tm, m.tra_tuples,
    rule=res_transmission_input_by_capacity_rule,
    doc='transmission input <= total transmission capacity')
```

```
def res_transmission_input_by_capacity_rule(m, tm, stf, sin, sout, tra,␣
↪com):
    return (m.e_tra_in[tm, stf, sin, sout, tra, com] <=
            m.dt * m.cap_tra[stf, sin, sout, tra, com])
```

Transmission Capacity Limit Rule: The constraint transmission capacity limit rule limits the variable total transmission capacity κ_{yaf}. This constraint restricts a transmission f through an arc a in support timeframe y from having more total power output capacity than an upper bound and having less than a lower bound. The constraint states that the variable total transmission capacity κ_{yaf} must be greater than or equal to the parameter transmission capacity lower bound $\underline{K}_{yaf}$ and less than or equal to the parameter transmission capacity upper bound $\overline{K}_{yaf}$. The mathematical explanation of this rule is given in *Multinode optimization model*.

In script `transmission.py` the constraint transmission capacity limit rule is defined and calculated by the following code fragment:

```
m.res_transmission_capacity = pyomo.Constraint(
    m.tra_tuples,
    rule=res_transmission_capacity_rule,
    doc='transmission.cap-lo <= total transmission capacity <= '
        'transmission.cap-up')
```

```
def res_transmission_capacity_rule(m, stf, sin, sout, tra, com):
    return (m.transmission_dict['cap-lo'][(stf, sin, sout, tra, com)],
            m.cap_tra[stf, sin, sout, tra, com],
            m.transmission_dict['cap-up'][(stf, sin, sout, tra, com)])
```

Transmission Symmetry Rule: The constraint transmission symmetry rule defines the power capacities of incoming and outgoing arcs a, a' of a transmission f in support timeframe y. The constraint states that the power capacities κ_{af} of the incoming arc a and the complementary outgoing arc a' between two sites must be equal. The mathematical explanation of this rule is given in *Multinode optimization model*.

In script `transmission.py` the constraint transmission symmetry rule is defined and calculated by the following code fragment:

```
m.res_transmission_symmetry = pyomo.Constraint(
    m.tra_tuples,
    rule=res_transmission_symmetry_rule,
    doc='total transmission capacity must be symmetric in both directions')
```

```
def res_transmission_symmetry_rule(m, stf, sin, sout, tra, com):
    return m.cap_tra[stf, sin, sout, tra, com] == (m.cap_tra
                                                   [stf, sout, sin, tra,␣
↪com])
```

DCPF Transmission Constraints

The following constraints are included in the model if the optional DC Power Flow feature is activated.

DC Power Flow Rule: The constraint DC Power Flow rule defines the power flow of transmission lines, which are modelled with DCPF. This constraint states that the power flow on a transmission line is equal to the product of voltage angle differences of two connecting sites v_{out} and v_{in} and the admittance of the transmission line. This constraint is only applied to the transmission lines modelled with DCPF. The mathematical explanation of this rule is given in *Multinode optimization model*. In script `transmission.py` the constraint DC Power Flow Rule is defined and calculated by the following code fragment:

```
m.def_dc_power_flow = pyomo.Constraint(
    m.tm, m.tra_tuples_dc,
    rule=def_dc_power_flow_rule,
    doc='transmission output = (angle(in)-angle(out))/ 57.2958 '
        '* -1 *(-1/reactance) * (base voltage)^2')
```

```
def def_dc_power_flow_rule(m, tm, stf, sin, sout, tra, com):
    return (m.e_tra_in[tm, stf, sin, sout, tra, com] ==
            (m.voltage_angle[tm, stf, sin] - m.voltage_angle[tm, stf,␣
↪sout]) / 57.2958 * -1 *
            (-1 / m.transmission_dict['reactance'][(stf, sin, sout, tra,␣
↪com)])
            * m.transmission_dict['base_voltage'][(stf, sin, sout, tra,␣
↪com)]
            * m.transmission_dict['base_voltage'][(stf, sin, sout, tra,␣
↪com)])
```

DCPF Transmission Input By Capacity Rule: The constraint DCPF transmission input by capacity rule expands the constraint transmission input by capacity rule for transmission lines modelled with

DCPF. This constraint limits the variable transmission input commodity flow π^{in}_{yaft} of DCPF transmission lines also with a lower bound. This constraint prevents the absolute value of the transmission power from exceeding the possible power input capacity of the line especially when the transmission power can be negative. The constraint states that the additive inverse of variable transmission input commodity flow $-\pi^{\text{in}}_{yaft}$ must be less than or equal to the variable total transmission capacity κ_{yaf}, scaled by the size of the time steps :math: *Delta t*. This constraint is only applied to the tranmission lines modelled with DCPF. The mathematical explanation of this rule is given in *Multinode optimization model*.

In script `transmission.py` the constraint transmission input by capacity rule is defined and calculated by the following code fragment:

```
m.res_transmission_dc_input_by_capacity = pyomo.Constraint(
    m.tm, m.tra_tuples_dc,
    rule=res_transmission_dc_input_by_capacity_rule,
    doc='-dcpf transmission input <= total transmission capacity')
```

```
def res_transmission_dc_input_by_capacity_rule(m, tm, stf, sin, sout, tra,␣
↪com):
    return (- m.e_tra_in[tm, stf, sin, sout, tra, com] <=
            m.dt * m.cap_tra[stf, sin, sout, tra, com])
```

Voltage Angle Limit Rule: The constraint voltage angle limit rule limits the maximum and minimum difference of voltage angles θ_{yvt} of two sites v_{out} and v_{in} connected with a DCPF transmission line with the parameter voltage angle difference limit $\overline{dl}_{yaf}$. This constraint is only applied to the transmission lines modelled with DCPF. The mathematical explanation of this rule is given in *Multinode optimization model*. In script `transmission.py` the constraint voltage angle limit rule is defined and given by the following code fragment:

```
m.def_angle_limit = pyomo.Constraint(
        m.tm, m.tra_tuples_dc,
        rule=def_angle_limit_rule,
        doc='-angle limit < angle(in) - angle(out) < angle limit')
```

```
def def_angle_limit_rule(m, tm, stf, sin, sout, tra, com):
    return (- m.transmission_dict['difflimit'][(stf, sin, sout, tra, com)],
            (m.voltage_angle[tm, stf, sin] - m.voltage_angle[tm, stf,␣
↪sout]),
            m.transmission_dict['difflimit'][(stf, sin, sout, tra, com)])
```

Absolute Transmission Flow Constraints: The two absolute transmission flow constraints are included in the model to create the variable absolute value of transmission commodity flow $\pi^{\text{in}}_{yaft}{}'$. By limiting the negative $-\pi^{\text{in}}_{yaft}{}'$ and positive $\pi^{\text{in}}_{yaft}{}'$ of substitute variable ''e_tra_abs" with the variable π^{in}_{yaft} and minimizing the substitute value $\pi^{\text{in}}_{yaft}{}'$ the absolute value of transmission commodity flow is retrieved. These constraints are only applied to the transmission lines modelled with DCPF. The mathematical explanation of these rules are given in *Multinode optimization model*. In script `transmission.py` the constraint Absolute Transmission Flow Constraints are defined and given by the following code fragment:

```
m.e_tra_abs1 = pyomo.Constraint(
    m.tm, m.tra_tuples_dc,
    rule=e_tra_abs_rule1,
    doc='transmission dc input <= absolute transmission dc input')
m.e_tra_abs2 = pyomo.Constraint(
```

(continues on next page)

(continued from previous page)

```
    m.tm, m.tra_tuples_dc,
    rule=e_tra_abs_rule2,
    doc='-transmission dc input <= absolute transmission dc input')
```

```
def e_tra_abs_rule1(m, tm, stf, sin, sout, tra, com):
    return (m.e_tra_in[tm, stf, sin, sout, tra, com] <=
            m.e_tra_abs[tm, stf, sin, sout, tra, com])
```

```
def e_tra_abs_rule2(m, tm, stf, sin, sout, tra, com):
    return (-m.e_tra_in[tm, stf, sin, sout, tra, com] <=
            m.e_tra_abs[tm, stf, sin, sout, tra, com])
```

Transmission Symmetry Rule: The above mentioned constraint transmission symmetry rule is only applied to the transmission lines modelled with transport model if the DCPF is activated. Since the DCPF transmission lines do not include the complementary arcs, this constraint is ignored for these transmission lines. For this reason, the constraint is indexed with the transmission tuple set `m.tra_tuples_tp` if the DCPF is activated.

In script `transmission.py` the constraint transmission symmetry rule is defined as following if the DCPF is activated:

```
m.res_transmission_symmetry = pyomo.Constraint(
    m.tra_tuples_tp,
    rule=res_transmission_symmetry_rule,
    doc='total transmission capacity must be symmetric in both directions')
```

Storage Constraints

Storage State Rule: The constraint storage state rule is the main storage constraint and it defines the storage energy content of a storage s in a site v in support timeframe y at a timestep t. This constraint calculates the storage energy content at a timestep t by adding or subtracting differences, such as ingoing and outgoing energy, to/from a storage energy content at a previous timestep $t-1$ multiplied by 1 minus the self-discharge rate d_{yvs} (which is scaled exponentially with the timestep size δt). Here ingoing energy is given by the product of the variable storage input commodity flow $\epsilon^{\text{in}}_{yvst}$ and the parameter storage efficiency during charge e^{in}_{yvs}. Outgoing energy is given by the variable storage output commodity flow $\epsilon^{\text{out}}_{yvst}$ divided by the parameter storage efficiency during discharge e^{out}_{yvs}. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage state rule is defined and calculated by the following code fragment:

```
m.def_storage_state = pyomo.Constraint(
    m.tm, m.sto_tuples,
    rule=def_storage_state_rule,
    doc='storage[t] = (1 - selfdischarge) * storage[t-1] + input * eff_in -
↪ output / eff_out')
```

```
def def_storage_state_rule(m, t, stf, sit, sto, com):
    return (m.e_sto_con[t, stf, sit, sto, com] ==
            m.e_sto_con[t - 1, stf, sit, sto, com] *
            (1 - m.storage_dict['discharge']
```

(continues on next page)

(continued from previous page)

```
             [(stf, sit, sto, com)]) ** m.dt.value +
            m.e_sto_in[t, stf, sit, sto, com] *
            m.storage_dict['eff-in'][(stf, sit, sto, com)] -
            m.e_sto_out[t, stf, sit, sto, com] /
            m.storage_dict['eff-out'][(stf, sit, sto, com)])
```

Storage Power Rule: The constraint storage power rule defines the variable total storage power $\kappa^{\mathrm{p}}_{yvs}$. The variable total storage power is defined by the constraint as the sum of the parameter storage power installed K^{p}_{yvs} and the variable new storage power $\hat{\kappa}^{\mathrm{p}}_{yvs}$. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage power rule is defined and calculated by the following code fragment:

```
m.def_storage_power = pyomo.Constraint(
    m.sto_tuples,
    rule=def_storage_power_rule,
    doc='storage power = inst-cap + new power')
```

```
def def_storage_power_rule(m, stf, sit, sto, com):
    if m.mode['int']:
        if (sit, sto, com, stf) in m.inst_sto_tuples:
            if (min(m.stf), sit, sto, com) in m.sto_const_cap_p_dict:
                cap_sto_p = m.storage_dict['inst-cap-p'][
                    (min(m.stf), sit, sto, com)]
            else:
                cap_sto_p = (
                    sum(m.cap_sto_p_new[stf_built, sit, sto, com]
                        for stf_built in m.stf
                        if (sit, sto, com, stf_built, stf) in
                        m.operational_sto_tuples) +
                    m.storage_dict['inst-cap-p'][(min(m.stf), sit, sto,␣
↪com)])
        else:
            cap_sto_p = (
                sum(m.cap_sto_p_new[stf_built, sit, sto, com]
                    for stf_built in m.stf
                    if (sit, sto, com, stf_built, stf)
                    in m.operational_sto_tuples))
    else:
        if (stf, sit, sto, com) in m.sto_const_cap_p_dict:
            cap_sto_p = m.storage_dict['inst-cap-p'][(stf, sit, sto, com)]
        else:
            cap_sto_p = (m.cap_sto_p_new[stf, sit, sto, com] +
                         m.storage_dict['inst-cap-p'][(stf, sit, sto,␣
↪com)])

    return cap_sto_p
```

Storage Capacity Rule: The constraint storage capacity rule defines the variable total storage size $\kappa^{\mathrm{c}}_{yvs}$. The variable total storage size is defined by the constraint as the sum of the parameter storage content installed K^{c}_{yvs} and the variable new storage size $\hat{\kappa}^{\mathrm{c}}_{yvs}$. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage capacity rule is defined and calculated by the following code fragment:

```
m.def_storage_capacity = pyomo.Constraint(
    m.sto_tuples,
    rule=def_storage_capacity_rule,
    doc='storage capacity = inst-cap + new capacity')
```

```
def def_storage_capacity_rule(m, stf, sit, sto, com):
    if m.mode['int']:
        if (sit, sto, com, stf) in m.inst_sto_tuples:
            if (min(m.stf), sit, sto, com) in m.sto_const_cap_c_dict:
                cap_sto_c = m.storage_dict['inst-cap-c'][
                    (min(m.stf), sit, sto, com)]
            else:
                cap_sto_c = (
                    sum(m.cap_sto_c_new[stf_built, sit, sto, com]
                        for stf_built in m.stf
                        if (sit, sto, com, stf_built, stf) in
                        m.operational_sto_tuples) +
                    m.storage_dict['inst-cap-c'][(min(m.stf), sit, sto,␣
↪com)])
        else:
            cap_sto_c = (
                sum(m.cap_sto_c_new[stf_built, sit, sto, com]
                    for stf_built in m.stf
                    if (sit, sto, com, stf_built, stf) in
                    m.operational_sto_tuples))
    else:
        if (stf, sit, sto, com) in m.sto_const_cap_c_dict:
            cap_sto_c = m.storage_dict['inst-cap-c'][(stf, sit, sto, com)]
        else:
            cap_sto_c = (m.cap_sto_c_new[stf, sit, sto, com] +
                         m.storage_dict['inst-cap-c'][(stf, sit, sto,␣
↪com)])

    return cap_sto_c
```

Storage New Capacity Rule: The constraint storage new capacity rule defines the newly installed capacity of a storage $\hat{\kappa}^{\mathrm{c}}_{yvs}$. This variable is defined by the constraint as the product of the variable new storage size units β^{c}_{yvs} and the parameter storage new capacity block $K^{\mathrm{c,block}}_{yvs}$. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage capacity rule is defined and calculated by the following code fragment:

```
m.def_new_cap_sto_c = pyomo.Constraint(
    m.sto_block_c_tuples,
    rule=def_new_cap_sto_c_rule,
    doc='cap_sto_c_new = cap_sto_c_unit * c-block')
```

```
def def_new_cap_sto_c_rule(m, stf, sit, sto, com):
    return (m.cap_sto_c_new[stf, sit, sto, com] ==
            m.sto_cap_c_unit[stf, sit, sto, com] *
            m.sto_block_c_dict[stf, sit, sto, com])
```

Storage New Power Rule: The constraint storage new power rule defines the newly installed power of a storage $\hat{\kappa}^{\mathrm{p}}_{yvs}$. This variable is defined by the constraint as the product of the variable new power size units β^{p}_{yvs} and the parameter storage new power block $K^{\mathrm{p,block}}_{yvs}$. The mathematical explanation of this

rule is given in *Energy Storage*.

In script `storage.py` the constraint storage capacity rule is defined and calculated by the following code fragment:

```
m.def_new_cap_sto_p = pyomo.Constraint(
    m.sto_block_p_tuples,
    rule=def_new_cap_sto_p_rule,
    doc='cap_sto_p_new = cap_sto_p_unit * p-block')
```

```
def def_new_cap_sto_p_rule(m, stf, sit, sto, com):
    return (m.cap_sto_p_new[stf, sit, sto, com] ==
            m.sto_cap_p_unit[stf, sit, sto, com] *
            m.sto_block_p_dict[stf, sit, sto, com])
```

Storage Input By Power Rule: The constraint storage input by power rule limits the variable storage input commodity flow $\epsilon^{\text{in}}_{yvst}$. This constraint restricts a storage s in a site v and support timeframe y at a timestep t from having more input power than the storage power capacity. The constraint states that the variable $\epsilon^{\text{in}}_{yvst}$ must be less than or equal to the variable total storage power κ^{p}_{yvs}, scaled by the size of the time steps :math: *Delta t*. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage input by power rule is defined and calculated by the following code fragment:

```
m.res_storage_input_by_power = pyomo.Constraint(
    m.tm, m.sto_tuples,
    rule=res_storage_input_by_power_rule,
    doc='storage input <= storage power')
```

```
def res_storage_input_by_power_rule(m, t, stf, sit, sto, com):
    return (m.e_sto_in[t, stf, sit, sto, com] <= m.dt *
            m.cap_sto_p[stf, sit, sto, com])
```

Storage Output By Power Rule: The constraint storage output by power rule limits the variable storage output commodity flow $\epsilon^{\text{out}}_{yvst}$. This constraint restricts a storage s in a site v and support timeframe y at a timestep t from having more output power than the storage power capacity. The constraint states that the variable $\epsilon^{\text{out}}_{vst}$ must be less than or equal to the variable total storage power κ^{p}_{yvs}, scaled by the size of the time steps Δt. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage output by power rule is defined and calculated by the following code fragment:

```
m.res_storage_output_by_power = pyomo.Constraint(
    m.tm, m.sto_tuples,
    rule=res_storage_output_by_power_rule,
    doc='storage output <= storage power')
```

```
def res_storage_output_by_power_rule(m, t, stf, sit, sto, co):
    return (m.e_sto_out[t, stf, sit, sto, co] <= m.dt *
            m.cap_sto_p[stf, sit, sto, co])
```

Storage State By Capacity Rule: The constraint storage state by capacity rule limits the variable storage energy content $\epsilon^{\text{con}}_{yvst}$. This constraint restricts a storage s in a site v and support timeframe y at a timestep t from having more storage content than the storage content capacity. The constraint states that the variable $\epsilon^{\text{con}}_{yvst}$ must be less than or equal to the variable total storage size κ^{c}_{yvs}. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage state by capacity rule is defined and calculated by the following code fragment.

```
m.res_storage_state_by_capacity = pyomo.Constraint(
    m.t, m.sto_tuples,
    rule=res_storage_state_by_capacity_rule,
    doc='storage content <= storage capacity')
```

```
def res_storage_state_by_capacity_rule(m, t, stf, sit, sto, com):
    return (m.e_sto_con[t, stf, sit, sto, com] <=
            m.cap_sto_c[stf, sit, sto, com])
```

Storage Power Limit Rule: The constraint storage power limit rule limits the variable total storage power $\kappa^{\mathrm{p}}_{yvs}$. This contraint restricts a storage s in a site v and support timeframe y from having more total power output capacity than an upper bound and having less than a lower bound. The constraint states that the variable total storage power $\kappa^{\mathrm{p}}_{yvs}$ must be greater than or equal to the parameter storage power lower bound $\underline{K}^{\mathrm{p}}_{yvs}$ and less than or equal to the parameter storage power upper bound $\overline{K}^{\mathrm{p}}_{yvs}$. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage power limit rule is defined and calculated by the following code fragment:

```
m.res_storage_power = pyomo.Constraint(
    m.sto_tuples,
    rule=res_storage_power_rule,
    doc='storage.cap-lo-p <= storage power <= storage.cap-up-p')
```

```
def res_storage_power_rule(m, stf, sit, sto, com):
    return (m.storage_dict['cap-lo-p'][(stf, sit, sto, com)],
            m.cap_sto_p[stf, sit, sto, com],
            m.storage_dict['cap-up-p'][(stf, sit, sto, com)])
```

Storage Capacity Limit Rule: The constraint storage capacity limit rule limits the variable total storage size $\kappa^{\mathrm{c}}_{yvs}$. This constraint restricts a storage s in a site v and support timeframe y from having more total storage content capacity than an upper bound and having less than a lower bound. The constraint states that the variable total storage size $\kappa^{\mathrm{c}}_{yvs}$ must be greater than or equal to the parameter storage content lower bound $\underline{K}^{\mathrm{c}}_{yvs}$ and less than or equal to the parameter storage content upper bound $\overline{K}^{\mathrm{c}}_{yvs}$. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage capacity limit rule is defined and calculated by the following code fragment:

```
m.res_storage_capacity = pyomo.Constraint(
    m.sto_tuples,
    rule=res_storage_capacity_rule,
    doc='storage.cap-lo-c <= storage capacity <= storage.cap-up-c')
```

```
def res_storage_capacity_rule(m, stf, sit, sto, com):
    return (m.storage_dict['cap-lo-c'][(stf, sit, sto, com)],
            m.cap_sto_c[stf, sit, sto, com],
            m.storage_dict['cap-up-c'][(stf, sit, sto, com)])
```

Initial And Final Storage State Rule: The constraint initial and final storage state rule defines and restricts the variable storage energy content $\epsilon^{\mathrm{con}}_{yvst}$ of a storage s in a site v and support timeframe y at the initial timestep t_1 and at the final timestep t_N. There are two distinct cases:

1. The initial and final storage states are specified by a value of the parameter I_{yvs} between 0 and 1. 2. I_{yvs} is not specified (e.g. by setting it '#NV' in the input sheet). In this case the initial and final storage state are still equal but variable.

In case 1 the constraints are written in the following way:

Initial storage state: Initial storage represents the storage state in a storage at the beginning of the simulation. The variable storage energy content $\epsilon^{\text{con}}_{yvst}$ at the initial timestep t_1 is defined by this constraint. The constraint states that the variable $\epsilon^{\text{con}}_{vst_1}$ must be equal to the product of the parameters storage content installed K^{c}_{yvs} and initial and final state of charge I_{yvs}.

Final storage state: Final storage represents the storage state in a storage at the end of the simulation. The variable storage energy content $\epsilon^{\text{con}}_{yvst}$ at the final timestep t_N is restricted by this constraint. The constraint states that the variable $\epsilon^{\text{con}}_{yvst_N}$ must be greater than or equal to the product of the parameters storage content installed K^{c}_{yvs} and initial and final state of charge I_{yvs}. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint initial and final storage state rule is then defined and calculated by the following code fragment:

```
m.res_initial_and_final_storage_state = pyomo.Constraint(
    m.t, m.sto_init_bound_tuples,
    rule=res_initial_and_final_storage_state_rule,
    doc='storage content initial == and final >= storage.init * capacity')
```

In case 2 the constraint becomes a lot easier, since the initial and final state are simply compared to each other by the following inequality:

$$\forall v \in V, s \in S\colon\ \epsilon^{\text{con}}_{vst_1} \leq \epsilon^{\text{con}}_{vst_N}$$

In script `storage.py` the constraint initial and final storage state rule is then defined and calculated by the following code fragment:

```
m.res_initial_and_final_storage_state_var = pyomo.Constraint(
    m.t, m.sto_tuples - m.sto_init_bound_tuples,
    rule=res_initial_and_final_storage_state_var_rule,
    doc='storage content initial <= final, both variable')
```

Storage Energy to Power Ratio Rule: For certain type of storage technologies, the power and energy capacities cannot be independently sized but are dependent to each other. Hence, the constraint storage energy to power ratio rule sets a linear dependence between the capacities through a user-defined "energy to power ratio" $k^{\text{E/P}}_{yvs}$. It has to be noted that this constraint is only active for the storages with a positive value under the column "ep-ratio" in the input file, and when this value is not given, the power and energy capacities can be sized independently. The mathematical explanation of this rule is given in *Energy Storage*.

In script `storage.py` the constraint storage energy to power rule is then defined and calculated by the following code fragment:

```
m.def_storage_energy_power_ratio = pyomo.Constraint(
    m.sto_en_to_pow_tuples,
    rule=def_storage_energy_power_ratio_rule,
    doc='storage capacity = storage power * storage E2P ratio')
```

```
def def_storage_energy_power_ratio_rule(m, stf, sit, sto, com):
    return (m.cap_sto_c[stf, sit, sto, com] == m.cap_sto_p[stf, sit, sto,␣
↪com] *
            m.storage_dict['ep-ratio'][(stf, sit, sto, com)])
```

Cost Constraints

The variable total system cost ζ is calculated by the cost function. In cases of CO2-minimization the total system cost is constrained by the following expression:

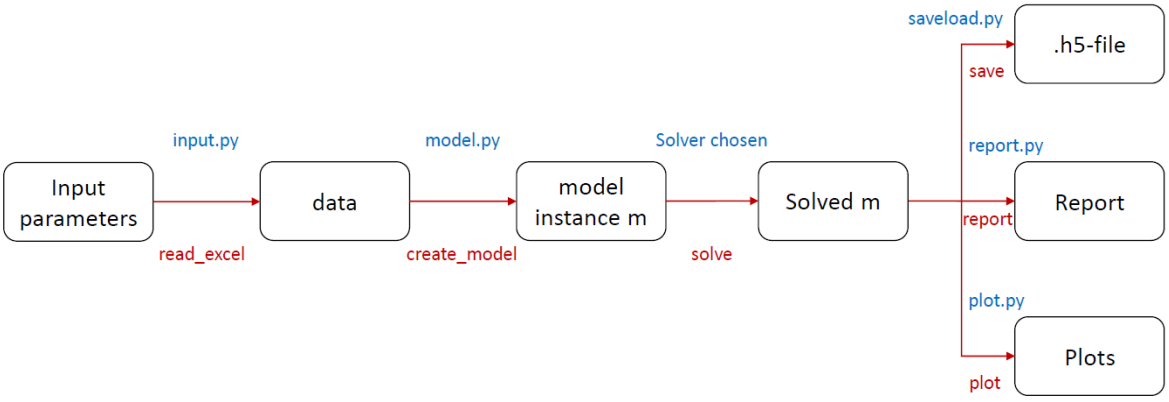
$$\zeta = \zeta_{\text{inv}} + \zeta_{\text{fix}} + \zeta_{\text{var}} + \zeta_{\text{fuel}} + \zeta_{\text{rev}} + \zeta_{\text{pur}} + \zeta_{\text{startup}} + \zeta_{\text{env}} \leq \overline{L}_{cost}$$

This constraint is given in `model.py` by the following code fragment.

```
def res_global_cost_limit_rule(m):
    if math.isinf(m.global_prop_dict["value"][min(m.stf), "Cost limit"]):
        return pyomo.Constraint.Skip
    elif m.global_prop_dict["value"][min(m.stf), "Cost limit"] >= 0:
        return(pyomo.summation(m.costs) <= m.global_prop_dict["value"]
               [min(m.stf), "Cost limit"])
    else:
        return pyomo.Constraint.Skip
```

1.3.2 'urbs' module description

This part gives a brief overview over the architecture of the program. The data flow in an urbs model is visualized in the following graph:



'urbs' uses a modular structure to build and execute the optimization and to automatically generate the results. All scripts are placed in the folder 'urbs'. In subfolder 'features' constraint expressions for the mathematical model are defined. These will not be discussed here and only the highest level functions will be discussed. The scripts used for these are the following (in alphabetical order):

identify.py

In this scripts the dictionary of input dataframes 'data' is parsed to conclude the structure of the problem to be built.

input.py

This file handles the input and prepares the mathematical model itself.

model.py

This file just includes the central function used for model generation.

output.py

This file contains lower level functions to retrieve data from a solved model instance.

plot.py

This script generates automated output pictures using the function

report.py

This script handles the automated generation of an excel data sheet from the solved model instance.

runfunctions.py

This file contains the central function for running a predefined set of inputs or a scenario thereof.

saveload.py

This file contains two functions to save and load a collection of inputs and the corresponding outputs of a model instance.

scenarios.py

In this script scenario functions are defined. These are used to automatically change the inputs as given in dictionary 'data'. In this way multiple runs of similar model instances can be automated.

validation.py

This file makes sure that the input given is not leading to an infeasible or non-sensical model. It generates error messages for certain known errors. It is a organically growing script.

1.4 ADMM module for regional decomposition

Continue here if you would like to use the regional decomposition module, using the alternating direction method of multipliers (ADMM).

1.4.1 ADMM module for regional decomposition

Overview

How to use the documentation

You should start with this overview which explains the underlying ideas of the regional decomposition. To fully comprehend the documentation you should be familiar with the urbs model already (see Package overview of the urbs documentation). Usually, when some content directly builds on a topic of the urbs documentation, this part of the documentation is explicitly referenced.

The *runscript explained* provides a detailed walkthrough of `runme_admm.py` and explains how to use decomposition for a model. It also explains the ADMM loop. After the overview you should continue with the tutorial to understand how to apply the code.

If you want to understand the mathematical background of ADMM, you should next look at the *Alternating direction method of multipliers (ADMM)*.

The implementation of the asynchronous ADMM method, can be seen in the section *Asynchronous ADMM implementation*. This section covers the workflow of the ADMM method, starting from the user-run runme_urbs script, the preparation script runfunctions_admm.py, the parallel jobs run_Worker.py and finally the Class urbsADMMmodel.

Finally the *ADMM user guide* gives ideas on how to improve, use, or extend the code, and on how to unify it with the urbs master branch.

Decomposition

First the concepts of decomposition are introduced. The idea of decomposition is that a large model might not fit into working memory, so it is desirable to split it into several smaller models that are independent to a certain degree. These models are called sub models. As the sub models are not truly independent there is a master model which coordinates the communication of the sub models.

Decomposition in energy system modelling

First the concepts of decomposition are introduced.

Due to computational and organizational reasons, it may be practical to partition an optimization problem (such as one depicting an energy system model) to multiple smaller "subproblems" . In case these smaller problems do not depend on each other, i.e. do not share any common (complicating) variables or common (complicating) constraints, then the approach is trivial: by solving these smaller problems independently, we recover the solution to the original problem.

However, energy systems consist often of subsystems which are dependent of each other. In this case, the solutions of the subproblems need to be consistent with each other, i.e. when one partitions an energy system model regionally, the interconnector power flows (and their capacities) between two regions would couple two regions. Mathematical decomposition methods are iterative methods, by which the subproblems are solved iteratively, and between each iteration, coordination steps are taken so that 1) the coupled subproblems are in consensus regarding their complicating variables and 2) the complicating constraints are satisfied. Thereby, these mettods offer here a so far not thoroughly tapped potential:

a) through the parallel solution of these subproblems, the large number of available processors can be made use of in order to overcome divergent runtimes

b) through sequential solution of these subproblems, only a subset of the original problem has to be contained in the working memory simultaneously.

Alternating direction method of multipliers (ADMM)

The decomposition methods implemented in this branch are based on the consensus variant of the alternating direction method of multipliers (ADMM).

Theoretical background of consensus ADMM

ADMM belongs to a family of decomposition methods based on dual decomposition. To understand its working principle, let's have a look at the following problem:

$$\begin{aligned} \min_{\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{y}_1,\boldsymbol{y}_2} \quad & f_1(\boldsymbol{x}_1,\boldsymbol{y}_1) + f_2(\boldsymbol{x}_2,\boldsymbol{y}_2) \\ \text{s.t.} \quad & \boldsymbol{x}_1 \in \chi_1, \quad \boldsymbol{x}_2 \in \chi_2 \end{aligned}$$

where indices $1, 2$ denote the first and second subsystems (e.g. two regions in an energy system model), with $\boldsymbol{x}_1, x_2$ as the sets of variables which are internal to the subsystems 1 and 2 respectively (e.g. set of generated power by processes in these subregions), $\boldsymbol{y}$ as the coupling variables between the subsystems 1 and 2 (e.g. the power flow between these subregions).

By creating two local copies of the complicating variable $(\boldsymbol{y}_1, \boldsymbol{y}_2)$ and introducing a "consensus" constraint which equates these to a global $\boldsymbol{y}_g$, this problem can be reformulated as follows:

$$\begin{aligned} \min_{\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{y}_1,\boldsymbol{y}_2} \quad & f_1(\boldsymbol{x}_1,\boldsymbol{y}_1) + f_2(\boldsymbol{x}_2,\boldsymbol{y}_2) \\ \text{s.t.} \quad & \boldsymbol{x}_1 \in \chi_1, \quad \boldsymbol{x}_2 \in \chi_2 \\ & \boldsymbol{y}_1 = \boldsymbol{y}_g \quad : \boldsymbol{\lambda}_1 \\ & \boldsymbol{y}_2 = \boldsymbol{y}_g \quad : \boldsymbol{\lambda}_2, \end{aligned}$$

where $\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2$, are the dual variables (Lagrange multipliers) of the two consensus constraints respectively. The augmented Lagrangian of such a problem looks as follows (with a set penalty parameter ρ):

$$\begin{aligned} & L(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{y}_1, \boldsymbol{y}_2, \boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2)_{\boldsymbol{x}_1 \in \chi_1, \boldsymbol{x}_2 \in \chi_2} \\ & = f_1(\boldsymbol{x}_1,\boldsymbol{y}_1) + f_2(\boldsymbol{x}_2,\boldsymbol{y}_2) + \boldsymbol{\lambda}_1^T(\boldsymbol{y}_1 - \boldsymbol{y}_g) + \boldsymbol{\lambda}_2^T(\boldsymbol{y}_2 - \boldsymbol{y}_g) + \frac{\rho}{2}\left\|\boldsymbol{y}_1 - \boldsymbol{y}_g\right\|_2^2 + \frac{\rho}{2}\left\|\boldsymbol{y}_2 - \boldsymbol{y}_g\right\|_2^2 \end{aligned}$$

From here, the essence of the consensus ADMM lies on decoupling this Lagrangian by fixing the global value and the Lagrangian multipliers which correspond to the consensus variables. For this, an arbitrary initialization can be made ($\boldsymbol{y}_g^0 := \boldsymbol{y}^0$, $\boldsymbol{\lambda}_{1,2}^0 := \boldsymbol{\lambda}^0$).

Then the following steps are applied for each iteration $\nu = \{1, \ldots, \nu_{\text{max}}\}$:

1) Through the fixing (or initialization, in case of the first step) of the global value and the Lagrangian multipliers, the decoupled models can be solved independently from each other:

$$(\boldsymbol{x}_1^{\nu+1}, \boldsymbol{y}_1^{\nu+1}) = \arg\min_{\boldsymbol{x}_1,\boldsymbol{y}_1} f_1(\boldsymbol{x}_1,\boldsymbol{y}_1) + (\boldsymbol{\lambda}_1^{\nu})^T(\boldsymbol{y}_1 - \boldsymbol{y}_g^{\nu}) + \frac{\rho}{2}\left\|\boldsymbol{y}_1 - \boldsymbol{y}_g^{\nu}\right\|_2^2 \text{ s.t. } \boldsymbol{x}_1 \in \chi_1$$

$$(\boldsymbol{x}_2^{\nu+1}, \boldsymbol{y}_2^{\nu+1}) = \arg\min_{\boldsymbol{x}_2,\boldsymbol{y}_2} f_1(\boldsymbol{x}_2,\boldsymbol{y}_2) + (\boldsymbol{\lambda}_2^{\nu})^T(\boldsymbol{y}_2 - \boldsymbol{y}_g^{\nu}) + \frac{\rho}{2}\left\|\boldsymbol{y}_2 - \boldsymbol{y}_g^{\nu}\right\|_2^2 \text{ s.t. } \boldsymbol{x}_2 \in \chi_2$$

2) Using these solutions, an averaging step is made to calculate the global value of the coupling variable to be used in the next iteration:

$$\boldsymbol{y}_g{}^{\nu+1} := (\boldsymbol{y}_1^{\nu+1} + \boldsymbol{y}_2^{\nu+1})/2$$

3) Then, the consensus Lagrangian multipliers need to be updated for each subproblem:

$$\boldsymbol{\lambda}_{1,2}^{\nu+1} := \boldsymbol{\lambda}_{1,2}^{\nu} + \rho \left(\boldsymbol{y}_{1,2}^{\nu+1} - \boldsymbol{y}_g{}^{\nu+1} \right)$$

4) Using the values obtained from 2) and 3), the primal and dual residuals are calculated for each subproblem:

$$r_{1,2}^{\nu+1} = \left\| \boldsymbol{y}_{1,2}^{\nu} - \boldsymbol{y}_g{}^{\nu} \right\|_2^2$$
$$d_{1,2}^{\nu+1} = \rho \left\| \boldsymbol{y}_g{}^{\nu+1} - \boldsymbol{y}_g{}^{\nu} \right\|_2^2$$

The steps 1, 2, and 3 and 4 are followed until convergence, which corresponds to the condition of primal and dual residuals being smaller than a user-set tolerance. For a more detailed description of consensus ADMM, please refer to the following material: https://stanford.edu/class/ee367/reading/admm_distr_stats.pdf.

Theoretical background of the asynchronous consensus ADMM

The consensus ADMM, whose steps were described above, is a synchronous algorithm. This means, each subproblem needs to be solved (step 1), in order for the updates (steps 2, 3) to take place before moving onto the next iteration. When the subproblems are solved in parallel for runtime benefits, this may lead to a so-called "straggler effect", where the performance of the algorithm is constrained by its slowest subproblem. This is often the case when the subproblems differ in sizes considerables (leading the small subproblems to have to wait for a larger problem to be solved).

In order to tackle this issue, an asynchronous variant of ADMM is formulated, where:

i) partial information from neighbours (a certain percentage η of the neighbors) is sufficient for each subproblem to move onto the next iteration, and

ii) the updating steps (2, 3) and the convergence checks take place locally rather than globally.

The specific algorithm is partially based on https://arxiv.org/abs/1710.08938. Here, a brief explanation of the algorithm will be made. For a more detailed description, please refer to this material.

Let us assume that our problem consists of the subsystems $k \in \{1, \ldots, \mathcal{N}\}$, with each subsystem k sharing some variable(s) with its neighbors $\mathcal{N}_k$. Asynchronicity takes places by each subproblem receiving the solutions from only up to $\lceil \eta \| \mathcal{N}_k \| \rceil$ neighbors before moving on to the next iteration. Since it takes different time for each of these subproblems to receive these information, each subproblem has their own iteration counters ν_k. A generalized notation of the problem variables are as follows:

Variable	Description
$\boldsymbol{x}_k$	Internal variables of subsystem k
$\boldsymbol{y}_{kl}$	Set of the coupling variables between subsystems k and l in subproblem k
$\boldsymbol{y}_k$	Set of the coupling variables between subsystems k and all its neighbors $\mathcal{N}_k$ in subproblem k
$\boldsymbol{y}_{g,kl}$	Set of the (now locally defined) global value of $\boldsymbol{y}_{kl}$ in subproblem k
$\boldsymbol{y}_{g,k}$	Set of the (now locally defined) global value of all coupling variables $\boldsymbol{y}_k$ in subproblem k
$\boldsymbol{\lambda}_{kl}$	Set of the Lagrange multipliers for the consensus constraint $\boldsymbol{y}_{kl} = \boldsymbol{y}_{g,kl}$ in the subproblem k
$\boldsymbol{\lambda}_k$	Set of the Lagrange multipliers for all consensus constraints $\boldsymbol{y}_k = \boldsymbol{y}_{g,k}$ in the subproblem k
ρ_k	Quadratic penalty parameter of the subproblem k

The asynchronous ADMM algorithm for each subsystem k operates as follows:

1) Through the fixing (or initialization, in case of the first step) of the global values and the Lagrangian multipliers, the decoupled model can be solved independently in parallel to the others:

$$(\boldsymbol{x}_k^{\nu_k+1}, \boldsymbol{y}_k^{\nu_k+1}) = \arg\min_{\boldsymbol{x}_k, \boldsymbol{y}_k} f_k(\boldsymbol{x}_k, \boldsymbol{y}_k) + (\boldsymbol{\lambda}_k^{\nu_k})^T(\boldsymbol{y}_k - \boldsymbol{y}_{g,k}^{\nu_k}) + \frac{\rho_k}{2}\left\|\boldsymbol{y}_k - \boldsymbol{y}_{g,k}^{\nu_k}\right\|_2^2 \text{ s.t. } \boldsymbol{x}_k \in \chi_k$$

2) Check if at least $\lceil \eta \|\mathcal{N}_k\| \rceil$ neighbors have new information to provide. If not, wait for it. If a problem l had already been solved multiple times since the last time information was received from it, pick the most recent information (corresponding to its current local iteration ν_l). (`recv()` is where this step is implemented):

3) For each neighbor l that provided new information, apply a modified averaging step (`update_z()` is where this step is implemented).

$$\forall l \in \mathcal{N}_k \text{ with new information: } \boldsymbol{y}_{g,kl}^{\nu+1} := \frac{\boldsymbol{\lambda}_{kl} + \boldsymbol{\lambda}_{lk} + \rho_k \boldsymbol{y}_{kl}^{\nu_k+1} + \rho_l \boldsymbol{y}_{lk}^{\nu_l}}{\rho_k + \rho_l}$$

This update step looks differently than that of synchronous ADMM, as it factors for the inaccuricies which arise from asynchronicity.

3) Update (all) consensus Lagrangian multipliers of subproblem k as usual:

$$\boldsymbol{\lambda}_k^{\nu_k+1} := \boldsymbol{\lambda}_k^{\nu_k} + \rho\left(\boldsymbol{y}_k^{\nu_k+1} - \boldsymbol{y}_{g,k}^{\nu_k+1}\right)$$

4) Update (all) consensus Lagrangian multipliers of subproblem k as usual:

$$\boldsymbol{\lambda}_k^{\nu_k+1} := \boldsymbol{\lambda}_k^{\nu_k} + \rho\left(\boldsymbol{y}_k^{\nu_k+1} - \boldsymbol{y}_{g,k}^{\nu_k+1}\right)$$

5) To check the convergence of a subproblem, collect all primal and dual residuals from the neighbors. If the maximum of these residuals is smaller than the convergence tolerance set for this subproblem, the subproblem converges:

$$r_{k,l}^{\nu+1} = \left\|\boldsymbol{y}_{kl}^{\nu} - \boldsymbol{y}_{g,kl}{}^{\nu}\right\|_2^2$$
$$d_{k,l}^{\nu+1} = \rho\ \left\|\boldsymbol{y}_g{}^{\nu+1} - \boldsymbol{y}_g{}^{\nu}\right\|_2^2$$

Interpretation of regional decomposition in urbs

In this implementation, the urbs model is regionally decomposed into "region clusters", where each model site can be clustered flexibly in separate subproblems. Drawing on the generic problem definition mentioned above, a specification of this notation can be made for urbs in the following way:

Variable	Description
$\boldsymbol{x}_k$	Process/storage capacities, throughputs, commodity flows:. within the region cluster k
$\boldsymbol{y}_{kl}$	Power flows/capacities of transmissions between the region clusters k and l (`e_tra_in(k,l)`, `cap_tra(k,l)`)

Formulation the global CO2 limit in the consensus form The intuition is that, when two region clusters are optimized separately, the coupling between them manifests itself in the transmission power flows and capacities between these clusters. Thereby, they constitute the complicating variables of the problem and hence the linear and quadratic consensus terms will have to be added to the respective cost functions. However, a simplification is made here, by ignoring the transmission capacities in the consensus variables. This simplifies the algorithm without having an influence on the feasibility of the solution, since

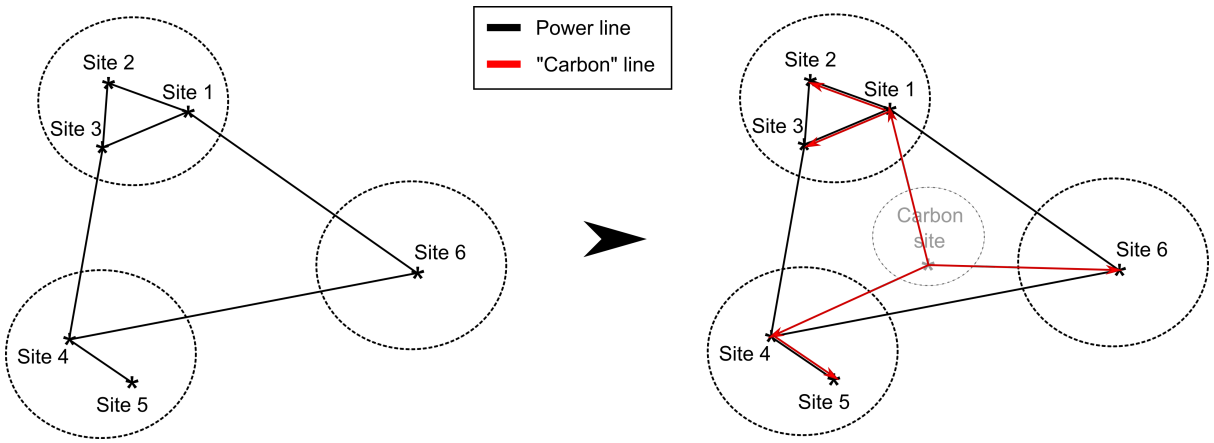
when the consensus for the power flows for a transmission line is achieved, the capacity of this transmission line will be set for each subproblem as the largest flow passing through this line to minimize the costs. In other words, the consensus of the power flows ensures the consensus of the line capacities.

Formulation the global CO2 limit in the consensus form

However, the line flows are not the sole coupling aspect in the urbs model. The global CO2 constraint, which restricts the total CO2 emissions produced by all of the regions, also couple the operation of the subproblem with each other. While this is a coupling constraint (and not a coupling variable), a reformulation into a similar consensus form can be made in the following way:

- A "dummy" region cluster (consisting of a single region) called `Carbon site` is created,
- A new stock commodity `Carbon` is created, which can be created in `Carbon site` for free, with a `max` amount equal to the global CO2 limit,
- The `Carbon` commodity act as "carbon certificates", such that to each process that emit `CO2`, it is added as an additional input commodity with an input ratio same as the output ratio of `CO2`,
- The `Carbon` commodity created in the `Carbon site` can be transported to each other sites for free. Therefore, new transmission "lines" are defined for this commodity, with unlimited capacity and no costs.

Now, the commodity flows of `Carbon` can be treated as an intercluster coupling variable (just like the power flows) and, as long as the consensus is achieved, the global CO2 limit will be respected.



Asynchronous ADMM implementation

This section explains the implementation of the asynchronous ADMM module. The workflow of the asynchronous ADMM module is established in the following way:

runme_admm.py: `runme_admm.py` is the script that has to be run by the user, where the input file for the model, modelled time period and the cluster definition is made.

runfunctions_admm.py: `runfunctions_admm.py` is the script that is called by the `runme_admm.py` script. Here, the data structures for the subproblems is created, the submodels are built, and asynchronous ADMM processes are launched.

run_Worker.py: `ADMM_async/run_Worker.py` includes the function `run_worker()`, which is the parallel ADMM routine that are followed asynchronously by the parallel workers. The major argu-

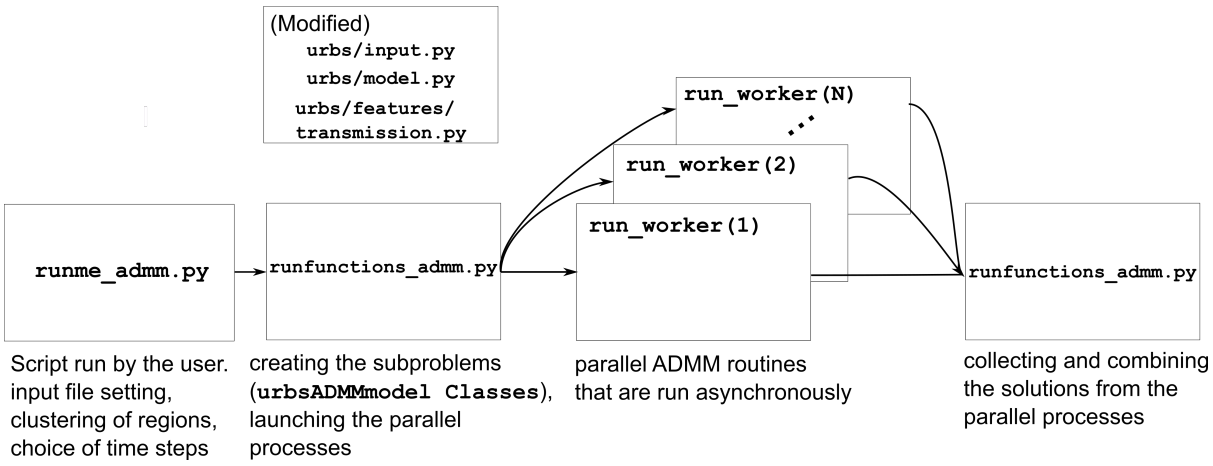
ment of this function is a `urbsADMMmodel` class, whose methods are defined in the `ADMM_async/urbs_admm_model.py` script.

Moreover, minor additions/modifications were done on the following, already existing scripts:

- urbs/input.py
- urbs/model.py
- urbs/features/transmission.py

which will also be mentioned here.

The workflow of the ADMM implementation is illustrated as follows:



In the following, a walkthrough on the scripts involved will be given to establish understanding regarding how the ADMM implementation works.

runme_admm.py

Let us start with the imported packages:

```
import os
import shutil
import urbs
from urbs.runfunctions_admm import *
from multiprocessing import freeze_support
```

Besides the usual urbs imports `os`, `shutil` and `urbs`, the `urbs.runfunctions` module is imported as it contains the `urbs.run_regional` function that commences the ADMM routine. Moreover, to allow for parallel operation on Windows systems, the `freeze_support` function has to be imported from the `multiprocessing` package.

Moving on to the input settings:

The script starts with the specification of the input file, which is to be located in the same folder as script `runme_admm.py`:

```
# Choose input file
input_files = 'germany.xlsx'  # for single year file name, for␣
↪intertemporal folder name
input_dir = 'Input'
input_path = os.path.join(input_dir, input_files)
```

Then the result name and the result directory is set:

```
result_name = 'Run'
result_dir = urbs.prepare_result_directory(result_name)  # name + time␣
↪stamp
```

Input file is added in the result directory:

```
# copy input file to result directory
try:
    shutil.copytree(input_path, os.path.join(result_dir, input_dir))
except NotADirectoryError:
    shutil.copyfile(input_path, os.path.join(result_dir, input_files))
# copy run file to result directory
shutil.copy(__file__, result_dir)
```

The objective function to be minimized by the model is then determined (options: 'cost' or 'CO2'):

```
# objective function
objective = 'cost'  # set either 'cost' or 'CO2' as objective
```

Then the specification of time step length and modeled time horizon is made:

```
# simulation timesteps
(offset, length) = (0, 8760)  # time step selection
timesteps = range(offset, offset+length+1)
dt = 1  # length of each time step (unit: hours)
```

Variable `timesteps` is the list of time steps to be modelled. Its members must be a subset of the labels used in `input_file`'s sheets "Demand" and "SupIm". It is one of the function arguments to `create_model()` and accessible directly, so that one can quickly reduce the problem size by reducing the simulation `length`, i.e. the number of time steps to be optimised. Finally, the variable `dt` gives the width of each timestep, input in hours.

`range()` is used to create a list of consecutive integers. The argument `+1` is needed, because `range(a,b)` only includes integers from `a` to `b-1`:

```
>>> range(1,11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

An essential input for the ADMM module is the clustering scheme of the model regions:

```
clusters = [[('Schleswig-Holstein')],[('Hamburg')],[('Mecklenburg-
↪Vorpommern')],[('Offshore')],[('Lower Saxony')],[('Bremen')],[('Saxony-
↪Anhalt')],[('Brandenburg')],[('Berlin')],[('North Rhine-Westphalia')],
            [('Baden-Württemberg')],[('Hesse')],[('Bavaria')],[('Rhineland-
↪Palatinate')],[('Saarland')],[('Saxony')],[('Thuringia')]]
```

The variable `clusters` is a list of tuples lists, where each element consists of tuple lists with the regions to be included in each subproblem. For instance, whereas the clustering given above yields each federal state of the Germany model having their own subproblems, a scheme as following:

```
clusters = [[('Schleswig-Holstein'),('Hamburg'),('Mecklenburg-Vorpommern'),
↪('Offshore'),('Lower Saxony'),('Bremen'),('Saxony-Anhalt'),('Brandenburg
↪'),('Berlin'),('North Rhine-Westphalia')],
            [('Baden-Württemberg'),('Hesse'),('Bavaria'),('Rhineland-
↪Palatinate'),('Saarland'),('Saxony'),('Thuringia')]]
```

(continues on next page)

(continued from previous page)

would yield two subproblems, where the northern and southern federal states of Germany are grouped with each other.

Then the color schemes for output plots is defined:

```
# add or change plot colors
my_colors = {
    'South': (230, 200, 200),
    'Mid': (200, 230, 200),
    'North': (200, 200, 230)}
for country, color in my_colors.items():
    urbs.COLORS[country] = color
```

Scenarios to be run can be then selected:

```
# select scenarios to be run
test_scenarios = [
    urbs.scenario_base
    ]
```

Finally, the `urbs.run_regional` function is called, commencing the ADMM routine:

```
if __name__ == '__main__':
    freeze_support()
    for scenario in test_scenarios:
        timesteps = range(offset, offset + length + 1)
        prob = urbs.run_regional(input_path, timesteps,
                                 scenario,result_dir,dt,objective,
                                 clusters=clusters)
```

To read about the `urbs.run_regional` function, please proceed to the next section, where the `runfunctions_admm.py` script, where this function resides, is described.

runfunctions_admm.py

Imports:

```
from pyomo.environ import SolverFactory
from .model import create_model
from .plot import *
from .input import *
from .validation import *
import urbs
import pandas as pd

import multiprocessing as mp
import queue
from .ADMM_async.run_Worker import run_worker
from .ADMM_async.urbs_admm_model import urbsADMMmodel
import time
import numpy as np
from math import ceil
```

Besides the usual imports of `runfunctions.py` (first group), additional imports are necessary:

- `multiprocessing` is a package that supports spawning processes using an API similar to the threading module. This is used for creating the objects `mp.Manager().Queue()` and `mp.Process()`.
- `queue` is used as an exception handling (`queue.Empty`), see later.
- The function `run_worker` contains all the ADMM steps that are followed by the submodel classes `urbsADMMmodel`.
- `time` is used as a runtime-profiling (for test purposes).
- `numpy` and `math.ceil` are required for array operations and a ceiling function respectively.

Some auxiliary function definitions follow:

```
def calculate_neighbor_cluster_per_line(boundarying_lines, cluster_idx,␣
↪clusters):
    neighbor_cluster = 99 * np.ones((len(boundarying_lines[cluster_idx]),␣
↪2))
    row_number = 0
    for (year, site_in, site_out, tra, com) in boundarying_lines[cluster_
↪idx].index:
        for cluster in clusters:
            if site_in in cluster:
                neighbor_cluster[row_number, 0] = clusters.index(cluster)
            if site_out in cluster:
                neighbor_cluster[row_number, 1] = clusters.index(cluster)
        row_number = row_number + 1
    cluster_from = neighbor_cluster[:, 0]
    cluster_to = neighbor_cluster[:, 1]
    neighbor_cluster = np.sum(neighbor_cluster, 1) - cluster_idx
    return cluster_from, cluster_to, neighbor_cluster
```

Function `calculate_neighbor_cluster_per_line` is applied to each cluster, and returns three arrays:

- `cluster_from` has a length equal to the boundarying transmission lines of a given cluster, and each value corresponds to the cluster index, that includes the site that is on the `sit_in` column of the transmission line,
- `cluster_to` has a length equal to the boundarying transmission lines of a given cluster, and each value corresponds to the cluster index, that includes the site that is on the `sit_out` column of the transmission line,
- `neighbor_cluster` has a length equal to the boundarying transmission lines of a given cluster, and each value corresponds to the index of the neighboring cluster that is involved.

```
def create_queues(clusters, boundarying_lines):
    edges = np.empty((1, 2))
    for cluster_idx in range(0, len(clusters)):
        edges = np.concatenate((edges, np.stack([boundarying_lines[cluster_
↪idx].cluster_from.to_numpy(),
                                                 boundarying_lines[cluster_
↪idx].cluster_to.to_numpy()], axis=1)))
    edges = np.delete(edges, 0, axis=0)
    edges = np.unique(edges, axis=0)
    edges = np.array(list({tuple(sorted(item)) for item in edges}))
```

(continues on next page)

(continued from previous page)

```
    queues = {}
    for edge in edges.tolist():
        fend = mp.Manager().Queue()
        tend = mp.Manager().Queue()
        if edge[0] not in queues:
            queues[edge[0]] = {}
        queues[edge[0]][edge[1]] = fend
        if edge[1] not in queues:
            queues[edge[1]] = {}
        queues[edge[1]][edge[0]] = tend
    return edges, queues
```

Function `create_queues` returns two objects:

- `edges` is an array with two columns, which expresses the connectivity between the clusters (if clusters are connected in the following way: `0--1--2`, `edges` would look as follows: `[[0, 1], [1, 0], [1, 2], [2, 1]]`),
- `queues` is a dictionary of dictionaries populated with `mp.Manager().Queue()` objects. There are as many `mp.Manager().Queue()` objects as the rows of `edges`, and these queues are used for the unidirectional data transfer between these clusters during the parallel operation.

Class `CouplingVars` is defined to store some coupling parameters:

```
class CouplingVars:
    flow_global = {}
    rhos = {}
    lambdas = {}
    cap_global = {}
    residdual = {}
    residprim = {}
```

Functions `prepare_result_directory` and `setup_solver` are unchanged except enforcing the barrier method for the gurobi solver (`method=2`). Please note that only gurobi is supported as a solver in this implementation!:

```
def prepare_result_directory(result_name):
    """ create a time stamped directory within the result folder.

    Args:
        result_name: user specified result name

    Returns:
        a subfolder in the result folder

    """
    # timestamp for result directory
    now = datetime.now().strftime('%Y%m%dT%H%M')

    # create result directory if not existent
    result_dir = os.path.join('result', '{}-{}'.format(result_name, now))
    if not os.path.exists(result_dir):
        os.makedirs(result_dir)

    return result_dir
```

(continues on next page)

(continued from previous page)

```
def setup_solver(optim, logfile='solver.log'):
    """ """
    if optim.name == 'gurobi':
        # reference with list of option names
        # http://www.gurobi.com/documentation/5.6/reference-manual/
↪parameters
        optim.set_options("logfile={}".format(logfile))
        optim.set_options("method=2")
        # optim.set_options("timelimit=7200")  # seconds
        # optim.set_options("mipgap=5e-4")  # default = 1e-4
    elif optim.name == 'glpk':
        # reference with list of options
        # execute 'glpsol --help'
        optim.set_options("log={}".format(logfile))
        # optim.set_options("tmlim=7200")  # seconds
        # optim.set_options("mipgap=.0005")
    elif optim.name == 'cplex':
        optim.set_options("log={}".format(logfile))
    else:
        print("Warning from setup_solver: no options set for solver "
              "'{}'!".format(optim.name))
    return optim
```

Now that the auxiliary functions are explained, the main function of this script, `run_regional`, will be explained step by step.

The docstring of the function gives an overview regarding the input and output arguments:

```
def run_regional(input_file, timesteps, scenario, result_dir,
                 dt, objective, clusters=None):
    """ run an urbs model for given input, time steps and scenario with␣
↪regional decomposition using ADMM

    Args:
        input_file: filename to an Excel spreadsheet for urbs.read_excel
        timesteps: a list of timesteps, e.g. range(0,8761)
        scenario: a scenario function that modifies the input data dict
        result_dir: directory name for result spreadsheet and plots
        dt: width of a time step in hours(default: 1)
        objective: the entity which is optimized ('cost' of 'co2')
        clusters: user-defined region clusters for regional decomposition␣
↪(list of tuple lists)

    Returns:
        the urbs model instances
    """
```

First, the model year is hard-coded to be used as the support year (`stf`) indices. This is a single scalar, since ADMM, in its current status, does not support intertemporal models:

```
# hard-coded year. ADMM doesn't work with intertemporal models (yet)
year = date.today().year
```

Then, similarly to regular urbs, the scenario is set up, the model data is read and and validations are

made in the following steps:

```
# scenario name, read and modify data for scenario
sce = scenario.__name__
data_all = read_input(input_file, year)
data_all = scenario(data_all)
validate_input(data_all)
validate_dc_objective(data_all, objective)
```

If there is a global CO2 limit set in the model, the necessary modifications to the data structure are made with the `add_carbon_supplier` function. These are mentioned in the section *Formulation the global CO2 limit in the consensus form*. Then, the *Carbon site* is added as a separate cluster:

```
if not data_all['global_prop'].loc[year].loc['CO2 limit', 'value'] == np.
↪inf:
    data_all = add_carbon_supplier(data_all, clusters)
    clusters.append(['Carbon_site'])
```

A *CouplingVars class* is initialized:

```
# initiate a coupling-variables Class
coup_vars = CouplingVars()
```

In the following code section, the `Transmission` DataFrame is sliced for each cluster (with index `cluster_idx`), such that `boundarying_lines[cluster_idx]` comprises only the transmission lines which are interfacing with a neighboring cluster and, conversely, `internal_lines[cluster_idx]` consists of the transmission lines that connect the sites within the cluster. Afterwards, the ADMM parameters `coup_vars.lambdas`, `coup_vars.rhos` and `coup_vars.flow_global` are initialized with the following indices:

- `cluster_idx`: each cluster index,
- `j`: each modelled time-step,
- `year`: the support timeframe (a single year in this case),
- `sit_from`: first end of the transmission line (obtained from `boundarying_lines[cluster_idx]`)
- `sit_to`: second end of the transmission line (obtained from `boundarying_lines[cluster_idx]`)

```
# identify the boundarying and internal lines
boundarying_lines = {}
internal_lines = {}

boundarying_lines_logic = np.zeros((len(clusters),
                                    data_all['transmission'].shape[0]),
                                   dtype=bool)
internal_lines_logic = np.zeros((len(clusters),
                                 data_all['transmission'].shape[0]),
                                dtype=bool)

for cluster_idx in range(0, len(clusters)):
    for j in range(0, data_all['transmission'].shape[0]):
        boundarying_lines_logic[cluster_idx, j] = (
                (data_all['transmission'].index.get_level_values('Site In
↪')[j]
```

(continues on next page)

(continued from previous page)

```
                in clusters[cluster_idx])
               ^ (data_all['transmission'].index.get_level_values('Site␣
↪Out')[j]
                  in clusters[cluster_idx]))
        internal_lines_logic[cluster_idx, j] = (
               (data_all['transmission'].index.get_level_values('Site In
↪')[j]
                in clusters[cluster_idx])
               and (data_all['transmission'].index.get_level_values('Site␣
↪Out')[j]
                    in clusters[cluster_idx]))

    boundarying_lines[cluster_idx] = \
        data_all['transmission'].loc[boundarying_lines_logic[cluster_idx,␣
↪:]]
    internal_lines[cluster_idx] = \
        data_all['transmission'].loc[internal_lines_logic[cluster_idx, :]]

    for i in range(0, boundarying_lines[cluster_idx].shape[0]):
        sit_from = boundarying_lines[cluster_idx].iloc[i].name[1]
        sit_to = boundarying_lines[cluster_idx].iloc[i].name[2]

        for j in timesteps[1:]:
            coup_vars.lambdas[cluster_idx, j, year, sit_from, sit_to] = 0
            coup_vars.rhos[cluster_idx, j, year, sit_from, sit_to] = 5
            coup_vars.flow_global[cluster_idx, j, year, sit_from, sit_to]␣
↪= 0
```

In the following optional step, the original problem is built and solved. This is the same as the regular urbs routine, and is used for testing purposes (e.g. comparing the ADMM result against this, making a runtime test). For your actual usage, feel free to comment this section out:

```
# (optional) create the central problem to compare results
prob = create_model(data_all, timesteps, dt, type='normal')

# refresh time stamp string and create filename for logfile
log_filename = os.path.join(result_dir, '{}.log').format(sce)

# setup solver
solver_name = 'gurobi'
optim = SolverFactory(solver_name)  # cplex, glpk, gurobi, ...
optim = setup_solver(optim, logfile=log_filename)

# original problem solution (not necessary for ADMM, to compare results)
orig_time_before_solve = time.time()
results_prob = optim.solve(prob, tee=False)
orig_time_after_solve = time.time()
orig_duration = orig_time_after_solve - orig_time_before_solve
flows_from_original_problem = dict((name, entity.value) for (name, entity)␣
↪in prob.e_tra_in.items())
flows_from_original_problem = pd.DataFrame.from_dict(flows_from_original_
↪problem, orient='index',
                                                     columns=['Original'])
```

In the next code section, `problems`, a list of `urbsADMMmodel` Classes and `sub`, a dictionary for keeping the Pyomo object of subproblems are initialized. Next, the following steps take place for each

region cluster `cluster_idx`:

- `problem` which is an instance of the `urbsADMMmodel` class, is initialized (please see the urbsADMMmodel, init Section),
- a Pyomo object for the subproblem is created using the `urbs.create_model` function with the `type='sub'` option, See the modified create_model in the model.py changes). This Pyomo instance is stored in the attribute `sub_pyomo` of `problem`,
- initial values for the global coupling variable values are stored in `problem.flow_global`, which is a subset of `coup_vars.flow_global` where the `cluster_idx` corresponds to the cluster in question,
- initial values for the consensus dual variables are stored in `problem.lamda`, which is a subset of `coup_vars.lambdas` where the `cluster_idx` corresponds to the cluster in question,
- initial value for the quadratic penalty parameter is stored in `problem.rho`,
- the unique index of the cluster is stored in `problem.ID`,
- the result directory and the scenario name are stored in the `problem.result_dir` and `problem.sce` respectively,
- the `cluster_from`, `cluster_to` and `neighbor_cluster` columns are appended to `boundarying_lines[cluster_idx]` DataFrame using the `calculate_neighbor_cluster_per_line` function. The appended DataFrame is then stored in `problem.boundarying_lines`
- the information for the total number of clusters is stored in `problem.na`
- the prepared instance `problem` is added to the list of `problems`

```
problems = []
sub = {}

# initiate urbs_admm_model Classes for each subproblem
for cluster_idx in range(0, len(clusters)):
    problem = urbsADMMmodel()
    sub[cluster_idx] = urbs.create_model(data_all, timesteps,␣
↪type='sub',
                                         sites=clusters[cluster_
↪idx],
                                         coup_vars=coup_vars,
                                         data_transmission_
↪boun=boundarying_lines[cluster_idx],
                                         data_transmission_
↪int=internal_lines[cluster_idx],
                                         cluster=cluster_idx)
    problem.sub_pyomo = sub[cluster_idx]
    problem.flow_global = {(key[1], key[2], key[3], key[4]): value
                           for (key, value) in coup_vars.flow_
↪global.items() if key[0] == cluster_idx}
    problem.flow_global = pd.Series(problem.flow_global)
    problem.flow_global.rename_axis(['t', 'stf', 'sit', 'sit_'],␣
↪inplace=True)
    problem.flow_global = problem.flow_global.to_frame()

    problem.lamda = {(key[1], key[2], key[3], key[4]): value
                     for (key, value) in coup_vars.lambdas.
↪items() if key[0] == cluster_idx}
```

(continues on next page)

(continued from previous page)

```
    problem.lamda = pd.Series(problem.lamda)
    problem.lamda.rename_axis(['t', 'stf', 'sit', 'sit_'],␣
↪inplace=True)
    problem.lamda = problem.lamda.to_frame()

    problem.rho = 5

    problem.ID = cluster_idx
    problem.result_dir = result_dir
    problem.sce = sce
    boundarying_lines[cluster_idx]['cluster_from'], boundarying_
↪lines[cluster_idx]['cluster_to'], \
        boundarying_lines[cluster_idx]['neighbor_cluster'] =␣
↪calculate_neighbor_cluster_per_line(boundarying_lines,
↪                                    cluster_idx,
↪                                    clusters)
    problem.boundarying_lines = boundarying_lines[cluster_idx]
    problem.na = len(clusters)
    problems.append(problem)
```

In the next step, `queues` are created for each communication channel using the `create_queues` *function*. These are then stored in the respective `problem`, along with the following attributes:

- `neighbors`: the indices of clusters that neighbor the cluster in question,
- `nneighbors`: the number of neighboring clusters,
- `nwait`: the number of neighboring subproblems, that the subproblem has to wait for in order to move on to the next iteration. This is calculated using the product `admmopt.nwaitPercent` of `nneighbors`, rounded up.

```
edges, queues = create_queues(clusters, boundarying_lines)

# define further necessary fields for the subproblems
for cluster_idx in range(0, len(clusters)):
    problems[cluster_idx].neighbors = sorted(set(boundarying_lines[cluster_
↪idx].neighbor_cluster.to_list()))
    problems[cluster_idx].nneighbors = len(problems[cluster_idx].neighbors)

    problems[cluster_idx].queues = dict((key, value) for (key, value) in␣
↪queues.items() if key == cluster_idx)
    problems[cluster_idx].queues.update(dict(
        (key0, {key: value}) for (key0, n) in queues.items() for (key,␣
↪value) in n.items() if
        key == cluster_idx).items())
    problems[cluster_idx].nwait = ceil(
        problems[cluster_idx].nneighbors * problems[cluster_idx].admmopt.
↪nwaitPercent)
```

Then, another Queue is created, which is used by each subproblem after they converge to send their solutions:

```
# define a Queue class for collecting the results from each subproblem␣
↪after convergence
```

(continues on next page)

(continued from previous page)

```
output = mp.Manager().Queue()
```

Afterwards, a list (`proc`) is initialized, and populated by `mp.Process` which take the function `run_worker`, to be run for each cluster. The arguments here are:

- `cluster_idx + 1`: ordinality of the cluster,
- `problems[cluster_idx]`: the `urbsADMMmodel` instance corresponding to the cluster,
- `output`: the Queue to be used for sending the subproblem solution

The processes are then launched using the `.start()` method.:

```
# define the asynchronous jobs for ADMM routines
procs = []
for cluster_idx in range(0, len(clusters)):
    procs += [mp.Process(target=run_worker, args=(cluster_idx + 1,␣
↪problems[cluster_idx], output))]

start_time = time.time()
start_clock = time.clock()
for proc in procs:
    proc.start()
```

While the processes are running, attempts to fetch results from `output` is made in constant intervals (0.5 seconds by default), until all child processes are finished (`while liveprocs:`). A soon as this is the case, we return to the parent thread (`proc.join()`):

```
# collect results as the subproblems converge
results = []
while liveprocs:
    try:
        while 1:
            results.append(output.get(False))
    except queue.Empty:
        pass

    time.sleep(0.5)
    if not output.empty():
        continue

    liveprocs = [p for p in liveprocs if p.is_alive()]

for proc in procs:
    proc.join()
```

Finally, the subproblem results are recovered and compared against the original problem in the following code section:

```
# ------------get results --------------------------
ttime = time.time()
tclock = time.clock()
totaltime = ttime - start_time
clocktime = tclock - start_clock

results = sorted(results, key=lambda x: x[0])
```

(continues on next page)

(continued from previous page)

```
obj_total = 0
obj_cent = results_prob['Problem'][0]['Lower bound']

for cluster_idx in range(0, len(clusters)):
    if cluster_idx != results[cluster_idx][0]:
        print('Error: Result of worker %d not returned!' % (cluster_idx +␣
↪1,))
        break
    obj_total += results[cluster_idx][1]['cost']

gap = (obj_total - obj_cent) / obj_cent * 100
print('The convergence time for original problem is %f' % (orig_duration,))
print('The convergence time for ADMM is %f' % (totaltime,))
print('The convergence clock time is %f' % (clocktime,))
print('The objective function value is %f' % (obj_total,))
print('The central objective function value is %f' % (obj_cent,))
print('The gap in objective function is %f %%' % (gap,))
```

The `run_worker` function (ADMM_async/run_worker.py)

In this section, the steps followed by the function `run_worker` is explained. This function is run in parallel by each subproblem, and it consists of some initialization steps, ADMM iterations and post-convergence steps.

The function takes three input arguments:

- `ID`: ordinality of the cluster (1 for the first subproblem, 2 for the second etc.),
- `s`: the `urbsADMMmodel` instance corresponding to the cluster,
- `output`: the Queue to be used for sending the subproblem solution

Since ADMM is an iterative method, the subproblems are expected to be solved multiple times (in the order of 10's, possibly 100's), with slightly different parameters in each iteration. The pyomo model which defines the optimization problem, first needs to be converted into a lower-level problem formulation (ultimately a set of matrices and vectors), which may take a very long time. Therefore, it is more practical that this conversion step happens only once, and the adjustments between iterations are made on the low-level problem formulation. Pyomo supports the usage of persistent solver interfaces (https://pyomo.readthedocs.io/en/stable/advanced_topics/persistent_solvers.html) for Gurobi, which exactly serves this purpose. These instances are created from the pyomo object with the following code section, and stored in the `sub_persistent` attribute:

```
s.sub_persistent = SolverFactory('gurobi_persistent')
s.sub_persistent.set_instance(s.sub_pyomo, symbolic_solver_labels=False)
```

Afterwards, the solver parameters can be directly set on the persistent solver instance (`Method=2` for barrier method, `Thread=1` for allowing the usage of a single CPU):

```
s.sub_persistent.set_gurobi_param('Method', 2)
s.sub_persistent.set_gurobi_param('Threads', 1)
```

The `.unique()` method is applied to `.neighbor_cluster` attribute to retrieve the unique neighbors:

```
s.neighbor_clusters = s.boundarying_lines.neighbor_cluster.unique()
```

The local iteration counter `nu` is initialized, and the maximum number of iterations `maxit` is retrieved from the `admmopt` attribute of the subproblem:

```
nu = 0  # iteration count
maxit = s.admmopt.iterMaxlocal  # get maximum iteration
```

The convergence flag is initialized as `False`, the convergence gap as `10**8` and an array keeping track of the objective function value of the solutions as `np.zeros`:

```
s.flag = False
s.gapAll = [10 ** 8] * s.na
cost_history = np.zeros(maxit)
```

The absolute convergence tolerance is calculated by scaling `s.conv_rel` (user input for relative convergence tolerance, set in the `admmopt` attribute of the subproblem) with the number of the coupling variables in the subproblem (`len(s.flow_global)`, added 1 to ensure convergence for the subproblems without any coupling variables):

```
s.convergetol = s.conv_rel * (len(s.flow_global)+1) #  # convergence␣
↪criteria for maximum primal gap
```

Now, the local ADMM iterations take place:

```
while nu <= maxit-1 and not s.flag:
```

First, if any message from neighbors is received (if `s.recvmsg` is not empty), the global values of the coupling variables are updated (with the `.update_z` *method*), along with choosing the quadratic penalty value that corresponds to the maximum among all the neighbors (with the `.choose_max_rho` *method*):

```
if s.recvmsg:
    s.update_z()  # update global flows
    s.choose_max_rho()  # update choose max rho
```

Then, to prepare the model for the next run, the updated global values, consensus Lagrange multipliers and penalty parameters are set for the Gurobi instance of the subproblem. For these steps, the *methods* `.fix_flow_global`, `.fix_lambda` and `set_quad_cost` is applied respectively:

```
s.fix_flow_global()
s.fix_lambda()

if nu > 0:
    s.set_quad_cost(rhos_old)
```

Now the subproblem can be solved, using the `.solve_problem` *method*:

```
s.result = s.solve_problem()
```

After solving the problem, the optimal values of the coupling variables are extracted using the *method* `.retrieve_boundary_flows`. The output of this method are twofold:

- `s.flows_all`: a `pd.MultiIndex` containing all the coupling variables,

- `s.flows_with_neighbor`: a dictionary of `pd.MultiIndex``es , whose elements are subsets of ``flows_all` that are shared with a certain neighbor. For instance, for the subproblem with indexs 0, s.flows_with_neighbor[2] will return the values of all coupling variables for the flows between the cluster 0 and 2.

Additionally, the objective value of the optimum is saved in `cost_history`:

```
# retrieve
s.flows_all, s.flows_with_neighbor = s.retrieve_boundary_flows()
cost_history[nu] = s.sub_persistent._solver_model.objval
```

After obtaining the solutions, the consensus Lagrange multiplier and quadratic penalty parameter is updated with the *method* `.update_y` and the *method* `.update_rho` respectively:

```
rhos_old = s.rho
if s.recvmsg:  # not the initialization
    s.update_y()  # update lambda
    s.update_rho(nu)
```

Convergence is checked with the `.converge` *method*:

```
# check convergence
s.flag = s.converge()
```

At the last step of each iteration, the recvmsg cache is emptied. Afterwards, relevant *messages* are sent to every neighbor, and are received from neighbors with the `.send` *method* and the `.recv` *method* respectively. For receiving methods, an optional argument `pollrounds` can be given. This gives the number of queries made for each message reception per neighbor (default value is 5), and thereby ensures that the message received is as up-to-date as possible.:

```
s.recvmsg = {}  # clear the received messages

s.send()
s.recv(pollrounds=5)
```

The local iteration counter is updated before moving onto the next iteration:

```
nu += 1
```

When the algorithm converges, the final pyomo model of the subproblem and the corresponding solution is saved with the save function:

```
save(s.sub_pyomo, os.path.join(s.result_dir, '_{}_'.format(ID),'{}.h5'.
↪format(s.sce)))
```

Additionally, a dictionary consisting of the final objective value, the values of coupling variables and primal/dual residuals is created and put into the `Queue` called `output`:

```
output_package = {'cost': cost_history[nu - 1], 'coupling_flows': s.flow_
↪global,
                  'primal_residual': s.primalgap, 'dual_residual': s.
↪dualgap}
output.put((ID - 1, output_package))
```

The urbsADMMmodel Class (ADMM_async/urbs_admm_model.py)

In this section, the initialization attributes and methods of the `urbsADMMmodel` class will be explained. This class is the main argument of the parallel calls of the `run_worker` function, encapsulates the local urbs subproblem and implements the ADMM steps including solving the subproblem, sending and recieving data to/from neighbors, updating global values of the coupling variables, the consensus Lagrange multipliers and the quadratic penalty parameters.

While the order in which these ADMM steps are followed is listed in the previous section, here the steps themselves will be described.

Starting with the attributes list of an `urbsADMMmodel` instance:

```
class urbsADMMmodel(object):
    def __init__(self):
        # initialize all the fields
        self.boundarying_lines = None
        self.flows_all = None
        self.flows_with_neighbor = None
        self.flow_global = None
        self.sub_pyomo = None
        self.sub_persistent = None
        self.neighbors = None
        self.nneighbors = None
        self.nwait = None
        self.var = {'flow_global': None, 'rho': None}
        self.ID = None
        self.nbor = {}
        self.pipes = None
        self.queues = None
        self.admmopt = admmoption()
        self.recvmsg = {}
        self.primalgap = [9999]
        self.dualgap = [9999]
        self.gapAll = None
        self.rho = None
        self.lamda = None
```

These attributes are described as follows:

- `self.boundarying_lines`: A pd.MultiIntex, that is a subset of Transmission lines that connect this cluster with other clusters,
- `self.flows_all`: a `pd.MultiIndex` containing the optimized values of all the coupling variables (`Elec` and `Carbon` flows) after a subproblem solution
- `self.flows_with_neighbor`: a dictionary of `pd.MultiIndex``es , whose elements are subsets of ``flows_all` that are shared with a certain neighbor
- `self.flow_global`: a `pd.MultiIndex` containing the global values of all the coupling variables (`Elec` and `Carbon` flows)
- `self.sub_pyomo`: a `pyomo.environ.ConcreteModel` object that represents the subproblem
- `self.sub_persistent`: a `GurobiPersistent` object, a persistent solver interface on which the model adjustments are made
- `self.neighbors`: the indices of clusters that neighbor the cluster in question

- `self.nneighbors`: the number of neighboring clusters
- `self.nwait`: the number of neighboring subproblems, that the subproblem has to wait for in order to move on to the next iteration. This is calculated using the product `admmopt.nwaitPercent` of `nneighbors`, rounded up.
- `self.ID`: the subproblem ID. An integer starting from 0 (for the first subproblem).
- `self.queues`: a dictionary of dictionary of `mp.Manager().Queue()` objects, which has the cluster in question either as the receiving or the sending end
- `self.admmopt`: an instance of the `admmoption` class. These include the ADMM parameters, which can be modified by the user. They will be listed below.
- `self.recvmsg`: an instance of the `message` class. This class is sent and received between the workers, and its attributes will be listed below.
- `self.primalgap`: an array which extends which each iteration, and keeps track of the primal residual of the solution
- `self.dualgap`: an array which extends which each iteration, and keeps track of the dual residual of the solution
- `self.gapAll`: a list which includes: primal residual of the subproblem, along with the primal residuals of neighboring clusters
- `self.rho`: a real number which represents the current value of the quadratic penalty parameter
- `self.lamda`: a `pd.MultiIndex` containing the values of the current consensus Lagrange multipliers

Before explaining the methods of `urbsADMMmodel` class, let us have a look at the two auxiliary classes `admmoption` and `message`:

```
class admmoption(object):
    """ This class defines all the parameters to use in admm """

    def __init__(self):
        self.rho_max = 10  # upper bound for penalty rho
        self.tau_max = 1.5  # parameter for residual balancing of rho
        self.tau = 1.05  # multiplier for increasing rho
        self.zeta = 1  # parameter for residual balancing of rho
        self.theta = 0.99  # multiplier for determining whether to update␣
↪rho
        self.mu = 10  # multiplier for determining whether to update rho
        self.pollWaitingtime = 0.001  # waiting time of receiving from one␣
↪pipe
        self.nwaitPercent = 0.2  # waiting percentage of neighbors (0, 1]
        self.iterMaxlocal = 20  # local maximum iteration
        self.rho_update_nu = 50 # rho is updated only for the first 50␣
↪iterations
        self.conv_rel = 0.1 # the relative convergece tolerance, to be␣
↪multiplied with (len(s.flow_global)+1)
```

The `admmoption` class includes numerous parameters that specify the ADMM method, which can be set by the user:

- `self.rho_max`: A positive real number, that sets an upper bound for the quadratic penalty parameter (see `.update_rho` for its usage)

- `self.tau_max`: A positive real number, that sets an upper bound for the per-iteration modifier of the quadratic penalty parameter (see `.update_rho` for its usage)
- `self.tau`: A positive real number, that scales the quadratic penalty parameter up or down (see `.update_rho` for its usage)
- `self.zeta`: A positive real number, that is used for the residual balancing of the quadratic penalty parameter (not in use currently)
- `self.theta`: A positive real number, that is used for the residual balancing of the quadratic penalty parameter (not in use currently)
- `self.mu`: A positive real number, that is used for the scaling of the quadratic penalty parameter (see `.update_rho` for its usage)
- `self.pollWaitingtime`: A positive real number, which represents the waiting time for receiving a message from a neighbor (see `recv` for its usage)
- `self.nwaitPercent`: A real number within (0, 1], that gives the percentage of its neighbors that a subproblem needs to receive a message in order to move onto the next iteration (see line 258 of `runfunctions_admm.py` for its usage)
- `self.iterMaxlocal`: A positive integer, that sets the maximum number of local iterations (see line 25 of `run_Worker.py` for its usage)
- `self.rho_update_nu`: A positive integer, that sets the last iteration number where the quadratic penalty parameter is updated. After this iteration number, it will not be updated anymore (see `.update_rho` for its usage)
- `self.conv_rel`: A positive real number, that is multiplied with `(len(s.flow_global)+1)` to set the absolute convergence tolerance of a local subproblem

Moving onto the `message` class:

```
class message(object):
    """ This class defines the message region i sends to/receives from j ""
↪"

    def __init__(self):
        self.fID = 0  # source region ID
        self.tID = 0  # destination region ID
        self.fields = {
            'flow': None,
            'rho': None,
            'lambda': None,
            'convergeTable': None}

    def config(self, f, t, var_flow, var_rho, var_lambda, gapall):  #␣
↪AVall and var are local variables of f region
        self.fID = f
        self.tID = t

        self.fields['flow'] = var_flow
        self.fields['rho'] = var_rho
        self.fields['lambda'] = var_lambda
        self.fields['convergeTable'] = gapall
```

Instances of this class are the packets that are communicated between the workers and contain the following attributes:

- `fID`: the index of the sending subproblem
- `tID`: the index of the receiving subproblem
- `fields`: a dictionary which consists of the exchanged message (the local optimizing values of coupling variables `flow`, the local quadratic parameter value `rho`, the local consensus Lagrange multiplier `lambda` and the local primal residual `gapall`)

Now let us return to the class `urbsADMMmodel` and go through its methods.

`.solve_problem` takes the persistent solver interface and solves it with the options `save_results` and `load_solutions` as `False` to save runtime. `warmstart` is set as `True`, even though the barrier solver does not support this feature yet.:

```
def solve_problem(self):
    self.sub_persistent.solve(save_results=False, load_solutions=False,␣
↪warmstart=True)
```

Three following methods (`.fix_flow_global`, `.fix_lambda` and `.set_quad_cost`) interface with the pyomo object and persistent solver interface of the subproblem, and modify the cost function with the updated global values of the coupling variable, consensus Lagrange multiplier and the quadratic penalty parameter. Observe that in the model, `lamda` (consensus Lagrange multiplier) and `flow_global` (global value of the coupling variable) are defined as `Variables` whose values are then fixed in the model with the `.fix` method, whereas the quadratic penalty parameter `rho` is a real number.:

```
def fix_flow_global(self):
    for key in self.flow_global.index:
        if not isinstance(self.flow_global.loc[key], pd.core.series.
↪Series):
            self.sub_pyomo.flow_global[key].fix(self.flow_global.loc[key])
            self.sub_persistent.update_var(
                self.sub_pyomo.flow_global[key])
        else:
            self.sub_pyomo.flow_global[key].fix(self.flow_global.loc[key,␣
↪0])
            self.sub_persistent.update_var(
                self.sub_pyomo.flow_global[key])

def fix_lambda(self):
    for key in self.lamda.index:
        if not isinstance(self.lamda.loc[key], pd.core.series.Series):
            self.sub_pyomo.lamda[key].fix(self.lamda.loc[key])
            self.sub_persistent.update_var(self.sub_pyomo.lamda[key])
        else:
            self.sub_pyomo.lamda[key].fix(self.lamda.loc[key, 0])
            self.sub_persistent.update_var(self.sub_pyomo.lamda[key])

def set_quad_cost(self, rhos_old):
    quadratic_penalty_change = 0
    # Hard coded transmission name: 'hvac', commodity 'Elec' for␣
↪performance.
    # Caution, as these need to be adjusted if the transmission of other␣
↪commodities exists!
    for key in self.flow_global.index:
        if (key[2] == 'Carbon_site') or (key[3] == 'Carbon_site'):
            quadratic_penalty_change += 0.5 * (
```

(continues on next page)

(continued from previous page)

```
                    self.rho - rhos_old) * \
                                        (self.sub_pyomo.e_tra_in[
                                             key, 'CO2_line', 'Carbon'] -
                                         self.sub_pyomo.flow_global[key])␣
↪** 2
        else:
            quadratic_penalty_change += 0.5 * (
                    self.rho - rhos_old) * \
                                        (self.sub_pyomo.e_tra_in[key, 'hvac
↪', 'Elec'] -
                                         self.sub_pyomo.flow_global[key])␣
↪** 2

    old_expression = self.sub_persistent._pyomo_model.objective_function.
↪expr
    self.sub_persistent._pyomo_model.del_component('objective_function')
    self.sub_persistent._pyomo_model.add_component('objective_function',
                                                   pyomo.Objective(expr =␣
↪old_expression + quadratic_penalty_change,
                                                                   ␣

↪sense=pyomo.minimize))
    self.sub_persistent.set_objective(
        self.sub_persistent._pyomo_model.objective_function)
    self.sub_persistent._solver_model.update()
```

With the methods `send` and `recv`, the message transfer bwetween subproblems take place. Let us start with `send`:

```
def send(self):
    dest = self.queues[self.ID].keys()
    for k in dest:
        # prepare the message to be sent to neighbor k
        msg = message()
        msg.config(self.ID, k, self.flows_with_neighbor[k], self.rho,
                   self.lamda[self.lamda.index.isin(self.flows_with_
↪neighbor[k].index)],
                   self.gapAll)
        self.queues[self.ID][k].put(msg)
```

The `send` method prepares a `message` for each neighbor `k`, where only the subset of the coupling variable and Lagrange multiplier values which are relevant to this neighbor are sent (`self.flows_with_neighbor[k]` and `self.lamda[self.lamda.index.isin(self.flows_with_neighbor[k].index)]`). Additionally, the quadratic penalty parameter `self.rho` and the local residual gap `self.gapAll` is also communicated. These values are inserted into the message with the `.config` method, and the message is sent (put into the `Queue`) using the `.put` method.

Next, the `.recv` method:

```
def recv(self, pollrounds=5):
    twait = self.admmopt.pollWaitingtime
    dest = list(self.queues[self.ID].keys())
    recv_flag = [0] * self.nneighbors
    arrived = 0  # number of arrived neighbors
    pollround = 0
```

(continues on next page)

(continued from previous page)

```
    # keep receiving from nbor 1 to nbor K in round until nwait neighbors␣
↪arrived
    while arrived < self.nwait and pollround < pollrounds:
        for i in range(len(dest)):
            k = dest[i]
            while not self.queues[k][self.ID].empty():  # read from queue␣
↪until get the last message
                self.recvmsg[k] = self.queues[k][self.ID].
↪get(timeout=twait)
                recv_flag[i] = 1
                # print("Message received at %d from %d" % (self.ID, k))
        arrived = sum(recv_flag)
        pollround += 1
```

The `recv` method attempts to receive the `message` from at least `self.nwait` neighbors. Within the loop `for i in range(len(dest))`, the message-reception queue from each neighbor is queried (with the `.get` method) until the queue is empty (hence `while not self.queues[k][self.ID].empty()`). When the `arrived` counter is at least `self.nwait`, the `.recv` procedure finishes.

Then we come to the three methods that update the global values of the coupling variable (`.update_z`), consensus Lagrange multiplier (`.update_y`) and the quadratic penalty parameter (`.update_rho`). Note that these methods are used to obtain new values for these variables, and their application to the problem takes place afterwards with the methods `.fix_flow_global`, `.fix_lambda` and `.set_quad_cost` as explained earlier.

Starting with `update_z`:

```
def update_z(self):
    srcs = self.queues[self.ID].keys()
    flow_global_old = deepcopy(self.flow_global)
    for k in srcs:
        if k in self.recvmsg and self.recvmsg[k].tID == self.ID:  # target␣
↪is this Cluster
            nborvar = self.recvmsg[k].fields  # nborvar['flow'], nborvar[
↪'convergeTable']
            self.flow_global.loc[self.flow_global.index.isin(self.flows_
↪with_neighbor[k].index)] = \
                (self.lamda.loc[self.lamda.index.isin(self.flows_with_
↪neighbor[k].index)] +
                 nborvar['lambda'] + self.flows_with_neighbor[k] * self.
↪rho + nborvar['flow'] * nborvar['rho']) \
                / (self.rho + nborvar['rho'])
    self.dualgap += [self.rho * (np.sqrt(np.square(self.flow_global - flow_
↪global_old).sum(axis=0)[0]))]
```

For updating the global variable, a loop is made, checking for each source (neighboring cluster) whether a new message is present that is meant for the cluster in question (`self.recvmsg and self.recvmsg[k].tID == self.ID`), and if yes, the global variable is updated using the equation provided in the theoretical section of the documentation. After the global value is updated using information from all sending neighbors, the new value for the dual residual is also calculated.

After updating the global flow value, the Lagrange multiplier update can be made by the `update_y` method using the equation provided in the theoretical section of the documentation:

```
def update_y(self):
    self.lamda = self.lamda + self.rho * (self.flows_all.loc[:, [0]] -␣
↪self.flow_global)
```

Then the quadratic penalty parameter is updated by the `.update_rho` method and then replaced by the maximum quadratic penalty parameter across all neighbors by the `.choose_max_rho` method:

```
# update rho and primal gap locally
def update_rho(self, nu):
    self.primalgap += [np.sqrt(np.square(self.flows_all - self.flow_
↪global).sum(axis=0)[0])]
    # update rho (only in the first rho_iter_nu iterations)
    if nu <= self.admmopt.rho_update_nu:
        if self.primalgap[-1] > self.admmopt.mu * self.dualgap[-1]:
            self.rho = min(self.admmopt.rho_max, self.rho * self.admmopt.
↪tau)
        elif self.dualgap[-1] > self.admmopt.mu * self.primalgap[-1]:
            self.rho = min(self.rho / self.admmopt.tau, self.admmopt.rho_
↪max)
    # update local converge table
    self.gapAll[self.ID] = self.primalgap[-1]

#   # use the maximum rho among neighbors for local update
def choose_max_rho(self):
    srcs = self.recvmsg.keys()
    for k in srcs:
        rho_nbor = self.recvmsg[k].fields['rho']
        self.rho = maximum(self.rho, rho_nbor)  # pick the maximum one
```

Whether the quadratic penalty parameter has to increase or decrease depends on the relation between `primalgap` and `dualgap`, `admmopt.tau`, `admmopt.tau_max` `admmopt.rho_max` and `admmopt.tau_max`. Therefore, before updating `rho`, the primal residual `primalgap` is also calculated within this method. For a mathematical description of the `rho` update, please refer to page 20 of https://stanford.edu/class/ee367/reading/admm_distr_stats.pdf.

The convergence is checked with the method `.converge`:

```
def converge(self):
    # first update local converge table using received converge tables
    if self.recvmsg is not None:
        for k in self.recvmsg:
            table = self.recvmsg[k].fields['convergeTable']
            self.gapAll = list(map(min, zip(self.gapAll, table)))
    # check if all local primal gaps < tolerance
    if max(self.gapAll) < self.convergetol:
        return True
    else:
        return False
```

Here, a convergence table is updated (or created, in case the first iteration) which consists of the primal residuals of all the neighboring subproblems and the subproblem in question itself (`self.gapAll = list(map(min, zip(self.gapAll, table)))`). If all of these local primal residuals are smaller than the absolute tolerance (`max(self.gapAll) < self.convergetol`), the method returns a `True`, and `False` otherwise.

The last method defined for `urbsADMMmodel` is `retrieve_boundary_flows`:

```
def retrieve_boundary_flows(self):
    e_tra_in_per_neighbor = {}

    self.sub_persistent.load_vars(self.sub_pyomo.e_tra_in[:, :, :, :, :,␣
↪:])
    boundary_lines_pairs = self.boundarying_lines.reset_index().set_index([
↪'Site In', 'Site Out']).index
    e_tra_in_dict = {(tm, stf, sit_in, sit_out): v.value for (tm, stf, sit_
↪in, sit_out, tra, com), v in
                     self.sub_pyomo.e_tra_in.items() if ((sit_in, sit_out)␣
↪in boundary_lines_pairs)}

    e_tra_in_dict = pd.DataFrame(list(e_tra_in_dict.values()),
                                 index=pd.MultiIndex.from_tuples(e_tra_in_
↪dict.keys())).rename_axis(
        ['t', 'stf', 'sit', 'sit_'])

    for (tm, stf, sit_in, sit_out) in e_tra_in_dict.index:
        e_tra_in_dict.loc[(tm, stf, sit_in, sit_out), 'neighbor_cluster']␣
↪= self.boundarying_lines.reset_index(). \
            set_index(['support_timeframe', 'Site In', 'Site Out']).
↪loc[(stf, sit_in, sit_out), 'neighbor_cluster']

    for neighbor in self.neighbors:
        e_tra_in_per_neighbor[neighbor] = e_tra_in_dict.loc[e_tra_in_dict[
↪'neighbor_cluster'] == neighbor]
        e_tra_in_per_neighbor[neighbor].reset_index().set_index(['t', 'stf
↪', 'sit', 'sit_'], inplace=True)
        e_tra_in_per_neighbor[neighbor].drop('neighbor_cluster', axis=1,␣
↪inplace=True)

    return e_tra_in_dict, e_tra_in_per_neighbor
```

This method loads the optimized flow variables from the model solution (`self.sub_persistent.load_vars(self.sub_pyomo.e_tra_in[:, :, :, :, :, :])`), and then applies to it a series of `pd.DataFrame` operations to produce the necessary data structures.

Changes made in the `create_model` function (model.py)

In the ADMM implementation, several adjustments were made in the model creation, for the specific case of creating the subproblems. Therefore, the `create_model` function now takes several additional optional input arguments:

```
def create_model(data_all, timesteps=None, dt=1, objective='cost',␣
↪dual=False, type='normal', sites = None, coup_vars=None, data_
↪transmission_boun=None, data_transmission_int=None, cluster=None):
```

Here, the `type=='sub'` specifies the case of creating a subproblem, `sites` are the model regions contained by the given cluster, `coup_vars` are the initialized values of the global flow values, `data_transmission_boun` and `data_transmission_int` are the data sets of transmission lines which include the intercluster and internal lines that are present for the considered subproblem, `cluster` is the index of the considered subproblem.

In the following, only the changes made on the `create_model` function for the ADMM implementation are mentioned.

The model preperation function `pyomo_model_prep` takes the model `type` as an argument, and creates a subset of the whole data structure `data_all` which is then passed to `data`:

```
if type == 'sub':
    m, data = pyomo_model_prep(data_all, timesteps, sites, type,
                  pd.concat([data_transmission_boun,data_transmission_
↪int]))  # preparing pyomo model
```

Note: Changes made in the ‘‘pyomo_model_prep‘‘ function (input.py, line 185)

In case the model type is `sub`, the cross-sections of the whole data structure which contains the specificed `sites` are taken:

```
data = deepcopy(data_all)
m.timesteps = timesteps
data['site_all']=data_all['site']
if type =='sub':
    m.global_prop = data_all['global_prop'].drop('description', axis=1)
    data['site'] = data_all['site'].loc(axis=0)[:,sites]
    data['commodity'] = data_all['commodity'].loc(axis=0)[:,sites]
    data['process'] = data_all['process'].loc(axis=0)[:,sites]
    data['storage'] = data_all['storage'].loc(axis=0)[:,sites]
    if sites != ['Carbon_site']:
        data['demand'] = data_all['demand'][sites]
        data['supim']= data_all['supim'][sites]
    else:
        data['demand'] = pd.DataFrame()
        data['supim'] = pd.DataFrame()
    data['transmission'] = data_transmission
```

Returning to `create_model`, in case the model type is `sub`, the quadratic penalty parameter `rho` is specified as the value that corresponds to the cluster in question:

```
if m.type =='sub':
    rho = dict((key[1:],value) for key, value in coup_vars.rhos.items() if␣
↪key[0] == cluster)
```

which is then set as a `pyomo.environ.Parameter`, along with `flow_global` (global values of coupling variables) and `lamda` (consensus Lagrange multipliers) as ‘‘pyomo.environ.Variable‘‘s:

```
if type=='sub':
    m.flow_global = pyomo.Var(
        m.tm,m.stf,m.sit,m.sit,
        within=pyomo.Reals,
        doc='flow global in')
    m.lamda = pyomo.Var(
        m.tm,m.stf,m.sit,m.sit,
        within=pyomo.Reals,
        doc='lambda in')
    m.rho = pyomo.Param(
        m.tm,m.stf,m.sit,m.sit,
        initialize=rho,
        doc='rho in')
```

In ADMM, the objective function is adjusted by the linear and quadratic penalty terms. This is imple-

mented via the following lines:

```
def cost_rule(m):
    if m.type =='sub':
        return (pyomo.summation(m.costs) + sum(0.5 * m.rho[(tm, stf, sit_
↪in, sit_out)] *
                        (m.e_tra_in[(tm, stf, sit_in,sit_out, tra, com)]
                        -m.flow_global[(tm, stf, sit_in,sit_out)])**2
                        for tm in m.tm
                        for stf, sit_in, sit_out, tra, com in m.tra_tuples_
↪boun) + sum(m.lamda[(tm, stf, sit_in, sit_out)] *
                        (m.e_tra_in[(tm,stf, sit_in,sit_out,tra,com)]
                        -m.flow_global[(tm, stf, sit_in,sit_out)])
                        for tm in m.tm
                        for stf, sit_in, sit_out, tra, com in m.tra_tuples_
↪boun)       )
```

In urbs, the transmission line capacities are built twice (once in both directions). Therefore, a halving of the investment and fixed costs has to be made in the pre-processing part of the data input. However, when the subsystems are decomposed, we have to introduce a further halving of the intercluster transmission lines, so that we avoid both clusters having to pay for this line twice as this would disrupt the costs of the whole system Therefore, the system costs `m.costs` are also defined with a slight difference:

```
elif m.type == 'sub':
    m.def_costs = pyomo.Constraint(
            m.cost_type,
            rule=def_costs_rule_sub,
            doc='main cost function by cost type')
```

One can see that the cost rule differs in name (`def_costs_rule_sub`). In this adjusted rule, the transmission costs are called via the function `transmission_cost_sub` instead of `transmission_costs`. This function is located in `urbs/features/transmission.py` at line 429 (note the coefficients `0.5`)

```
def transmission_cost_sub(m, cost_type):
    """returns transmission cost function for the different cost types"""
    if cost_type == 'Invest':
        cost = (sum(m.cap_tra_new[t] *
                    m.transmission_dict['inv-cost'][t] *
                    m.transmission_dict['invcost-factor'][t]
                    for t in m.tra_tuples - m.tra_tuples_boun)
        + 0.5 * sum(m.cap_tra_new[t] *
                    m.transmission_dict['inv-cost'][t] *
                    m.transmission_dict['invcost-factor'][t]
                    for t in m.tra_tuples_boun))
        if m.mode['int']:
            cost -= (sum(m.cap_tra_new[t] *
                        m.transmission_dict['inv-cost'][t] *
                        m.transmission_dict['overpay-factor'][t]
                        for t in m.tra_tuples_internal)
            + 0.5 * sum(m.cap_tra_new[t] *
                        m.transmission_dict['inv-cost'][t] *
                        m.transmission_dict['overpay-factor'][t]
                        for t in m.tra_tuples_boun))
        return cost
    elif cost_type == 'Fixed':
```

(continues on next page)

(continued from previous page)

```
        return (sum(m.cap_tra[t] * m.transmission_dict['fix-cost'][t] *
                    m.transmission_dict['cost_factor'][t]
                    for t in m.tra_tuples_internal)
                + 0.5 * sum(m.cap_tra[t] * m.transmission_dict['fix-cost
↪'][t] *
                    m.transmission_dict['cost_factor'][t]
                    for t in m.tra_tuples_boun))
    elif cost_type == 'Variable':
        if m.mode['dpf']:
            return (sum(m.e_tra_in[(tm,) + t] * m.weight *
                        m.transmission_dict['var-cost'][t] *
                        m.transmission_dict['cost_factor'][t]
                        for tm in m.tm
                        for t in m.tra_tuples_tp) + \
                    sum(m.e_tra_abs[(tm,) + t] * m.weight *
                        m.transmission_dict['var-cost'][t] *
                        m.transmission_dict['cost_factor'][t]
                        for tm in m.tm
                        for t in m.tra_tuples_dc))
        else:
            return (sum(m.e_tra_in[(tm,) + t] * m.weight *
                        m.transmission_dict['var-cost'][t] *
                        m.transmission_dict['cost_factor'][t]
                        for tm in m.tm
                        for t in m.tra_tuples_internal)
                        + 0.5 * sum(m.e_tra_in[(tm,) + t] * m.weight *
                        m.transmission_dict['var-cost'][t] *
                        m.transmission_dict['cost_factor'][t]
                        for tm in m.tm
                        for t in m.tra_tuples_boun))
```

This concludes the documentation of the ADMM implementation on urbs.

ADMM user guide

This section serves as a guide for those who would like to use the regional decomposition module by ADMM.

Setting the modelled time steps

As with the usual urbs, the modelled time steps has to be set on `runme_admm.py` in the corresponding *line*

Clustering scheme for the regional decomposition

Regional decomposition only makes sense if the energy system model contains multiple sites. These sites then need to be assigned to different subproblems in "clusters", whose scheme has to be input on `runme_admm.py` within the variable `clusters` in the corresponding *line*:

```
clusters = [[('site 1 of cluster 1'),('site 2 of cluster 1'),('site 3 of␣
↪cluster 1')],
            [('site 1 of cluster 2'),('site 2 of cluster 2')]]
```

Any number of clusters is possible, from two to the total number of sites (each site forming its own cluster). For the trivial case of having only a single cluster, the regional decomposition is obviously not necessary.

The input of ADMM parameters

The initialized values of ADMM parameters can be set in the following *line* on the `runfunctions_admm.py` script:

```
for j in timesteps[1:]:
    coup_vars.lambdas[cluster_idx, j, year, sit_from, sit_to] = 0
    coup_vars.rhos[cluster_idx, j, year, sit_from, sit_to] = 5
    coup_vars.flow_global[cluster_idx, j, year, sit_from, sit_to] = 0
```

as well as *here* again for the quadratic penalty parameter:

```
problem.rho = 5
```

ADMM settings (`admmoption`)

Lastly, the ADMM settings, which are input as attributes of the class `admmoption` of `urbsADMMmodel` can be fine tuned depending on the problem type. These settings can be found in the *corresponding section* of `ADMM_async/urbs_admm_model.py`:

```
# ##--------ADMM parameters specification ---------------------------------
↪----
class admmoption(object):
    """ This class defines all the parameters to use in admm """

    def __init__(self):
        self.rho_max = 10  # upper bound for penalty rho
        self.tau_max = 1.5  # parameter for residual balancing of rho
        self.tau = 1.05  # multiplier for increasing rho
        self.zeta = 1  # parameter for residual balancing of rho
        self.theta = 0.99  # multiplier for determining whether to update␣
↪rho
        self.mu = 10  # multiplier for determining whether to update rho
        self.pollWaitingtime = 0.001  # waiting time of receiving from one␣
↪pipe
        self.nwaitPercent = 0.2  # waiting percentage of neighbors (0, 1]
        self.iterMaxlocal = 20  # local maximum iteration
        #self.convergetol = 365 * 10 ** 1#  convergence criteria for␣
↪maximum primal gap
        self.rho_update_nu = 50 # rho is updated only for the first 50␣
↪iterations
        self.conv_rel = 0.1 # the relative convergece tolerance, to be␣
↪multiplied with len(s.flow_global)
```

Commenting out the original problem solution

The `runfunctions_admm.py` includes the routines for building and solution of the original, undecomposed model for testing purposes. When the problem is solved in a decomposed way, the original problem doesn't need to be solved. Therefore, the *following code section* has to be commented out in actual operation:

```
# (optional) create the central problem to compare results
prob = create_model(data_all, timesteps, dt, type='normal')

# refresh time stamp string and create filename for logfile
log_filename = os.path.join(result_dir, '{}.log').format(sce)

# setup solver
solver_name = 'gurobi'
optim = SolverFactory(solver_name)  # cplex, glpk, gurobi, ...
optim = setup_solver(optim, logfile=log_filename)

# original problem solution (not necessary for ADMM, to compare results)
orig_time_before_solve = time.time()
results_prob = optim.solve(prob, tee=False)
orig_time_after_solve = time.time()
orig_duration = orig_time_after_solve - orig_time_before_solve
flows_from_original_problem = dict((name, entity.value) for (name, entity)␣
↪in prob.e_tra_in.items())
flows_from_original_problem = pd.DataFrame.from_dict(flows_from_original_
↪problem, orient='index',
                                                     columns=['Original'])
```

as well as the *test procedure* at the end of `runfunctions_admm.py`:

```
# ------------get results ---------------------------
ttime = time.time()
tclock = time.clock()
totaltime = ttime - start_time
clocktime = tclock - start_clock

results = sorted(results, key=lambda x: x[0])

obj_total = 0
obj_cent = results_prob['Problem'][0]['Lower bound']

for cluster_idx in range(0, len(clusters)):
    if cluster_idx != results[cluster_idx][0]:
        print('Error: Result of worker %d not returned!' % (cluster_idx +␣
↪1,))
        break
    obj_total += results[cluster_idx][1]['cost']

gap = (obj_total - obj_cent) / obj_cent * 100
print('The convergence time for original problem is %f' % (orig_duration,))
print('The convergence time for ADMM is %f' % (totaltime,))
print('The convergence clock time is %f' % (clocktime,))
print('The objective function value is %f' % (obj_total,))
print('The central objective function value is %f' % (obj_cent,))
print('The gap in objective function is %f %%' % (gap,))
```

CHAPTER 2

Features

- urbs is a linear programming optimization model for multi-commodity energy systems, their sizing, development and utilization.
- It finds the minimum cost energy system to satisfy given demand timeseries for possibly multiple commodities (e.g. electricity, heat).
- By default, operates on hourly-spaced timesteps (configurable) and can be used for intertemporal optimization.
- Thanks to pandas, complex data analysis code is short and extensible.
- The model itself is quite small thanks to relying on the Pyomo package.
- urbs includes reporting and plotting functions for rapid scenario development.

CHAPTER 3

Changes

3.1 2019-03-13 Version 1.0

- Maintenance: Modularity (only features which are used are build)
- Maintenance: New structure of documentation
- Feature: Time variable efficiency
- Feature: Objective function can be changed to CO2
- Feature: Intertemporal feature (expansion between years)
- Feature: Input validation (having easier to understand error messages due to Excel file)
- Feature: Reconstruction of partial feature
- Feature: Global constraints instead of Hacks
- Bugfixes: Many

3.2 2017-01-13 Version 0.7

- Maintenance: Model file `urbs.py` split into subfiles in folder `urbs`
- Feature: Usable area in site implemented as possible constraint
- Feature: Plot function (and `get_timeseries`) now support grouping of multiple sites
- Feature: Environmental commodity costs (e.g. emission taxes or other pollution externalities)
- Bugfix: column *Overproduction* in report sheet did not respect DSM

3.3 2016-08-18 Version 0.6

- *Demand Side Management Constraints* added
- *Process Constraints for partial operation* added
- Various fixes in examples, docs and tutorials for Pyomo 4/Python 3 changes

3.4 2016-02-16 Version 0.5

- Support for Python 3 added
- Support for Pyomo 4 added, while maintaining Pyomo 3 support. Upgrading to Pyomo 4 is advised, as support while be dropped with the next release to support new features.
- New feature: maximal power gradient for conversion processes
- Documentation: *buyselldoc* (expired) long explanation for *Buy* and *Sell* commodity types
- Documentation: *Model Implementation* full listing of sets, parameter, variables, objective function and constraints in mathematical notation and textual explanation
- Documentation: updated installation notes in README.md
- Plotting: automatic sorting of time series by variance makes it easier to read stacked plots with many technologies

3.5 2015-07-29 Version 0.4

- Additional commodity types *Buy* and *Sell*, which support time-dependent prices.
- Persistence functions *load* and *save*, based on pickle, allow saving and retrieving input data and problem instances including results, for later re-plotting or re-analysis without having to solve them again.
- Documenation: *workflow* tutorial added with example "Newsealand"

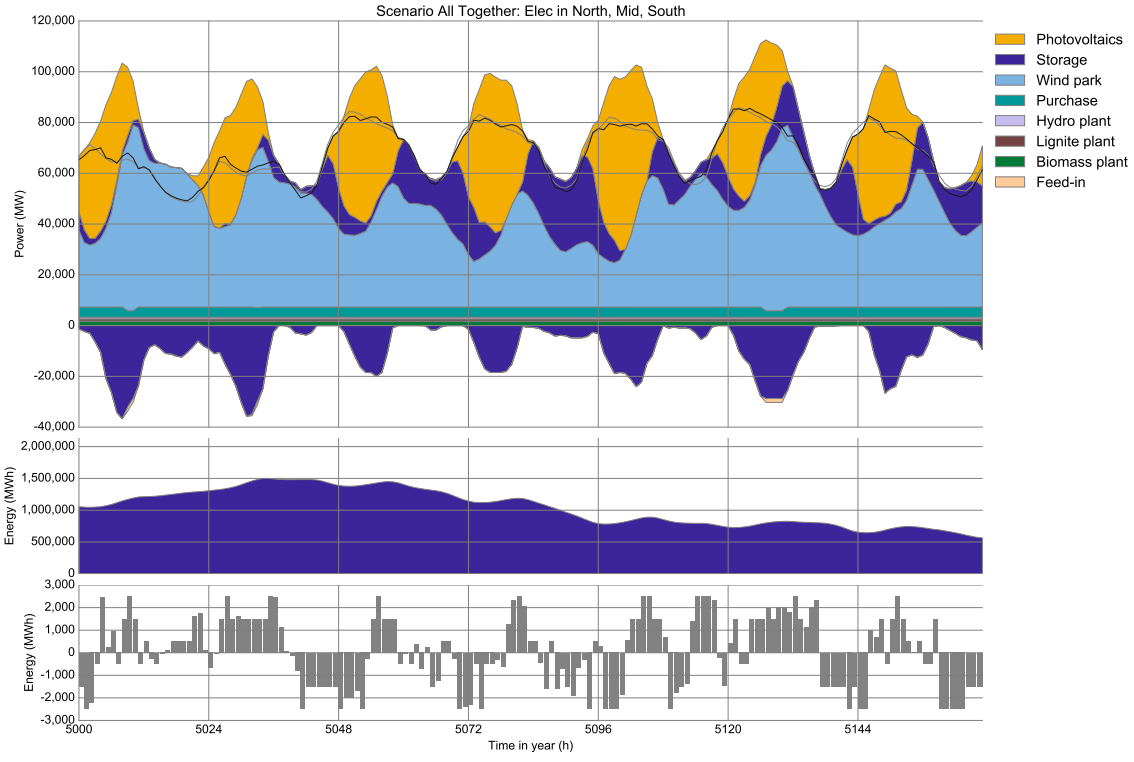
3.6 2014-12-05 Version 0.3

- Processes now support multiple inputs and multiple output commodities.
- As a consequence `plot()` now plots commodity balance by processes, not input commodities.
- urbs now supports input files with only a single site; simply delete all entries from the 'Transmission' spreadsheet and only use a single site name throughout your input.
- Moved hard-coded 'Global CO2 limit' constraint to dedicated "Hacks" spreadsheet, while the constraint is `add_hacks()`.
- More docstrings and comments in the main file `urbs.py`.

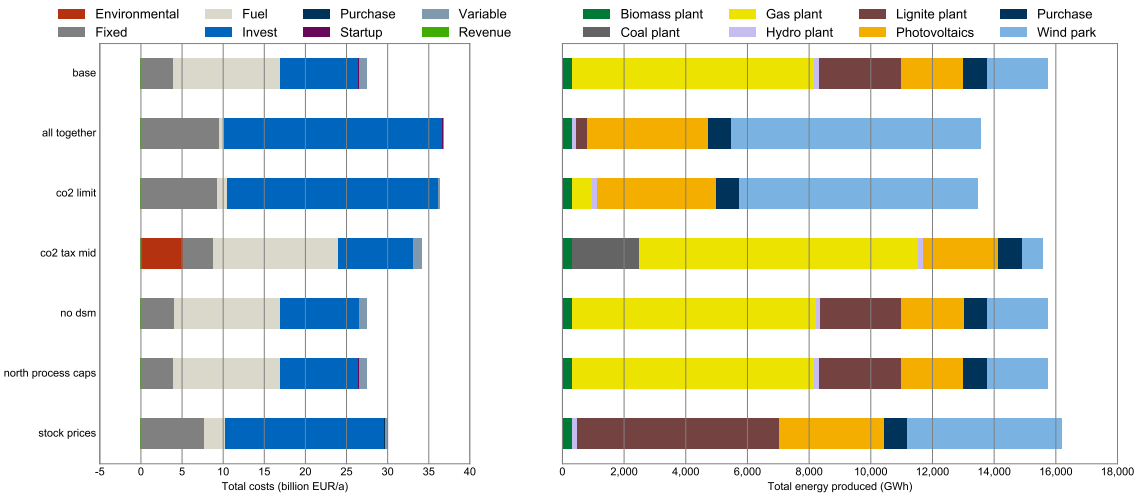
CHAPTER 4

Screenshots

This is a typical result plot created by `urbs.plot()`, showing electricity generation and storage levels in one site over 10 days (240 time steps):



An exemplary comparison script `comp.py` shows how one can create automated cross-scenario analyses with very few lines of pandas code. This resulting figure shows system costs and generated electricity by energy source over five scenarios:


Environmental
Fixed
Fuel
Invest
Purchase
Startup
Variable
Revenue
base
all together
co2 limit
co2 tax mid
no dsm
north process caps
stock prices
-5
0
5
10
15
20
25
30
35
40
Total costs (billion EUR/a)
Biomass plant
Coal plant
Gas plant
Hydro plant
Lignite plant
Photovoltaics
Purchase
Wind park
0
2,000
4,000
6,000
8,000
10,000
12,000
14,000
16,000
18,000
Total energy produced (GWh)

CHAPTER 5

Dependencies

- Python versions 2.7 or 3.x are both supported.
- pyomo for model equations and as the interface to optimisation solvers (CPLEX, GLPK, Gurobi, ...). Version 4 recommended, as version 3 support (a.k.a. as coopr.pyomo) will be dropped soon.
- matplotlib for plotting due to its capability to customise everything.
- pandas for input and result data handling, report generation
- Any solver supported by pyomo; suggestion: GLPK

Python Module Index

u

Index

C

U