urbs Documentation

Release 0.7

tum-ens

Oct 29, 2019

Contents

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

Organization Chair of Renewable and Sustainable Energy Systems, Technical University of Munich

Version 0.7

Date Oct 29, 2019

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 Overview

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 four 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.

Stock commodities have three numeric attributes that represent their price, total annual and per timestep supply. Environmental commodities (i.e. CO2) have a maximum allowed quantity that may be created.

Commodities are defined over the tuple `(site, commodity, type)`, for example `(Norway, wind, SupIm)` for wind in Norway with a time series or `(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 `(site, process)` specifies the location of a given process e.g. `(Iceland, turbine)` would locate a process `turbine` at site `Iceland`. The second tuple `(process, commodity, direction)` then specifies the inputs and outputs for that process. For example, `(turbine, geothermal, In)` and `(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 `(site in, site out, transmission, commodity)`. For example, `(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. A self-discharge term is **not** included at the moment, but could be added trivially (one column, one modification of the storage state equation). 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 `(site, storage, stored commodity)`. For example, `(Norway, pump storage, electricity)` represents a pump storage power plant in Norway that can store and retrieve energy in form of electricity.

Timeseries

Demand

Each combination `(site, demand commidty)` may have one timeseries, describing the (average) power demand (MWh/h) per timestep. They are a crucial input parameter, as the whole optimisation aims to satisfy these demands with minimal costs by the given technologies (process, storage, transmission).

Intermittent Supply

Each combination `(site, supim commodity)` must be supplied with one timeseries, normalised to a maximum value of 1 relative to the installed capacity of a process using this commodity as input. For example, a wind power timeseries should reach value 1, when the wind speed exceeds the modelled 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.

1.1.2 Tutorial

The README file contains installation notes. This tutorial expands on the steps that follow this installation.

This tutorial is a commented walk-through through the script `runme.py`, which is a demonstration user script that can serve as a good basis for one's own script.

Initialisation

Imports

```
try:
    import pyomo.environ
    from pyomo.opt.base import SolverFactory
    PYOMO3 = False
except ImportError:
    import coopr.environ
    from coopr.opt.base import SolverFactory
    PYOMO3 = True
import os
import shutil
import urbs
from datetime import datetime
```

Several packages are included.

- the try-except block checks for the version of Coopr/Pyomo installed and imports

the necessary packages for the model creation and solution.

- os is a builtin Python module, included here for its os.path submodule that offers operating system independent path manipulation routines. The following code creates the path string `'result/foo'` or `'result\\foo'` (depending on the operating system), checks whether it exists and creates the folder(s) if needed. This is used to prepare a new directory for generated result file:

  ```
  result_dir = os.path.join('result', 'foo')
  if not os.path.exists(result_dir):
      os.makedirs(result_dir)
  ```

- urbs is the module whose functions are used mainly in this script. These are `read_excel()`, `create_model()`, `report()` and `plot()`. More functions can be found in the document API reference.
- **‘pyomo.opt.base‘_** is a utility package by pyomo and provides the function `SolverFactory` that allows creating a `solver` object. This objects hides the differences in input/output formats among solvers from the user. More on that in section *Solving* below.
- *datetime* is used to append the current date and time to the result directory name (used in `prepare_result_directory()`)

Settings

From here on, the script is best read from the back.:

```
if __name__ == '__main__':
    input_file = 'mimo-example.xlsx'
    result_name = os.path.splitext(input_file)[0]  # cut away file␣
↪extension
    result_dir = prepare_result_directory(result_name)  # name + time stamp

    (offset, length) = (4000, 5*24)  # time step selection
    timesteps = range(offset, offset+length+1)
```

Variable `input_file` defines the input spreadsheet, from which the optimization problem will draw all its set/parameter data.

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 two function arguments to `create_model()` and accessible directly, because one can quickly reduce the problem size by reducing the simulation `length`, i.e. the number of timesteps to be optimised.

`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]
```

The following 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.

In addition to defining scenarios, the `scenarios` list allows to select a subset to be actually run.

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.

Scenario selection

```
# select scenarios to be run
scenarios = [
    scenario_base,
    scenario_stock_prices,
    scenario_co2_limit,
    scenario_north_process_caps,
    scenario_all_together]
scenarios = scenarios[:1]  # select by slicing
```

In Python, functions are objects, so they can be put into data structures just like any variable could be. In the following, the list `scenarios` is used to control which scenarios are being actually computed.

Run scenarios

```
for scenario in scenarios:
    prob = run_scenario(input_file, timesteps, scenario, result_dir)
```

Having prepared settings, input data and scenarios, the actual computations happen in the function `run_scenario()` of the script. 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 = urbs.read_excel(input_file)
data = scenario(data)
```

Function *`read_excel()`* returns a dict `data` of six 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.

Solving

```
# create model
prob = urbs.create_model(data, timesteps)
if PYOMO3:
    prob = prob.create()

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

(continues on next page)

(continued from previous page)

```
# solve model and read results
optim = SolverFactory('glpk')  # cplex, glpk, gurobi, ...
optim = setup_solver(optim, logfile=log_filename)
result = optim.solve(prob, tee=True)
if PYOMO3:
    prob.load(result)
else:
    prob.solutions.load_from(result)
```

This section is the "work horse", where most computation and time is spent. The optimization problem is first defined (*create_model()*), then filled with values (`create`). 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 two lines call the solver and read 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.

Reporting

```
# write report to spreadsheet
urbs.report(
    prob,
    os.path.join(result_dir, '{}-{}.xlsx').format(sce, now),
    ['Elec'], ['South', 'Mid', 'North'])
```

The *urbs.report()* function creates a spreadsheet from the main results. Summaries of costs, emissions, capacities (process, transmissions, storage) are saved to one sheet each. For timeseries, each combination of the given `sites` and `commodities` are summarised both in sum (in sheet "Energy sums") and as individual timeseries (in sheet "... timeseries"). See also *Reporting function explained* for a detailed explanation of this function's implementation.

Plotting

```
# 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
```

First, the use of the module constant *COLORS* for customising plot colors is demonstrated. All plot colors are user-defineable 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:
```

(continues on next page)

(continued from previous page)

```
    # create figure
    fig = urbs.plot(prob, com, sit)

    # change the figure title
    ax0 = fig.get_axes()[0]
    nice_sce_name = sce.replace('_', ' ').title()
    new_figure_title = ax0.get_title().replace(
        'Energy balance of ', '{}: '.format(nice_sce_name))
    ax0.set_title(new_figure_title)

    # save plot to files
    for ext in ['png', 'pdf']:
        fig_filename = os.path.join(
            result_dir, '{}-{}-{}-{}.{}').format(sce, com, sit, now, ext)
        fig.savefig(fig_filename, bbox_inches='tight')
```

Finally, for each demand commodity (only `Elec` in this case), a plot over the whole optimisation period is created. If `timesteps` were longer, a shorter plotting period could be defined and given as an optional argument to `plot()`.

The paragraph "change figure title" shows exemplarily how to use matplotlib manually to modify some aspects of a plot without having to recreate the plotting function from scratch. For more ideas for adaptations, look into `plot()`'s code or the matplotlib documentation.

The last paragraph uses the `savefig()` method to save the figure as a pixel `png` (raster) and `pdf` (vector) image. The `bbox_inches='tight'` argument eliminates whitespace around the plot.

Note: `savefig()` has some more interesting arguments. For example `dpi=600` can be used to create higher resolution raster output for use with printing, in case the preferable vector images cannot be used. The filename extension or the optional `format` argument can be used to set the output format. Available formats depend on the used plotting backend. Most backends support png, pdf, ps, eps and svg.

1.1.3 Workflow

This page is a step-by-step explanation on how to get one's own model running. For the sake of an example, assume you want to investigate whether the (imaginary) state *New Sealand* with its four islands *Vled Haven*, *Qlyph Archipelago*, *Stryworf Key*, and *Jepid Island* would benefit by linking their islands' power systems by costly underground cables to better integrate fluctuating wind power generation.

Prerequisites

You have followed the sections installation instructions and get started in the README, i.e. you can successfully execute an optimisation run with the example dataset `mimo-example.xlsx` with the example run script `runme.py`. These two files will serve as a scaffold for your own investigation.

Create a private development branch

Using git, create and directly checkout a new branch with a topical name. Good names should tell you the goal of a branch, so something along the lines of `test1234` is no good name. For this project, `newsealand` looks like a good name:

```
$ git checkout -b newsealand
```

The private branch can be used to commit your own changes, while benefitting from new features/bug fixes that are pushed to the master branch on GitHub. Whenever you want to retreive those new changes, execute the following commands:

```
$ git fetch origin
$ git rebase origin/master
```

A full explanation of how to use git is beyond the scope of this documentation, so please refer to the Git book, especially chapter 1, 2, 3.

Create an input data file

Create a copy of the file `mimo-example.xlsx` and give it short, descriptive name `newsealand.xlsx`. Open it.

Go through the sheets, either adding, deleting or modifying rows. Keep the column titles as they are, because they are required by the model. Each title has a tooltip that explains the use of the parameter.

If you have created a development branch, this is a good time to add this file to version control:

```
$ git add newsealand.xlsx
$ git commit -m "added newsealand.xlsx"
```

Site, DSM and Buy-Sell-price

Note at the outset, that you do not have to worry about the three mentioned worksheets, since they are not used for this tutorial. You need to keep them, however, and modify them in order to avoid problems. First, specify the four desired Sites in **Site** and set all values to either `NV()` or `inf`. In the sheet **DSM** enter the four islands of New Sealand as sites into the corresponding fields and set all values in the columns *cap-max-do* and *cap-max-up* to `0`. You do not need to change anything in sheet **Buy-Sell-Price**.

Commodity

Remove the rows with unneeded commodities, here everything except **Gas**, **Elec**, **Wind**, **CO2**, and **Slack**. *New Sealand* only uses these for power generation. While **Slack** is not needed, it makes debugging unexpected model behaviour much easier. Better keep it. Rename the sites to match the island names. The file should now contain 20 rows, 5 for each island.

Let's assume that *Jepid Island* does not have access to **Gas**, so change the parameter `max` and `maxperstep` to 0. Island *Stryworf Key* does have a gas connection, but the pipeline can only deliver 50 MW worth of Gas power.

These steps result in the following table. The bolded values result from the assumptions described in the previous paragraphs. The other values are left unchanged from the example dataset:

Table 1: Sheet **Commodity**; empty cells correspond to `=NV()` (*no value*) fields

Site	Commodity	Type	price	max	maxperstep
Jepid Island	CO2	Env		inf	inf
Jepid Island	Elec	Demand			
Jepid Island	Gas	Stock	27.0	**0.0**	**0.0**
Jepid Island	Slack	Stock	999.0	inf	inf
Jepid Island	Wind	SupIm			
Qlyph Archipelago	CO2	Env		inf	inf
Qlyph Archipelago	Elec	Demand			
Qlyph Archipelago	Gas	Stock	27.0	inf	inf
Qlyph Archipelago	Slack	Stock	999.0	inf	inf
Qlyph Archipelago	Wind	SupIm			
Stryworf Key	CO2	Env		inf	inf
Stryworf Key	Elec	Demand			
Stryworf Key	Gas	Stock	27.0	inf	**50.0**
Stryworf Key	Slack	Stock	999.0	inf	inf
Stryworf Key	Wind	SupIm			
Vled Haven	CO2	Env		inf	inf
Vled Haven	Elec	Demand			
Vled Haven	Gas	Stock	27.0	inf	inf
Vled Haven	Slack	Stock	999.0	inf	inf
Vled Haven	Wind	SupIm			

You have done some work already. It's time for another commit. Instead of adding every changed file manually, you can add option `-a` to the commit, which adds all **unstaged changes** from `git status` to the next commit. With that:

```
$ git commit -am "changed commodities to 4 islands in newsealand.xlsx"
```

Note: From now on, commit yourself whenever you reach a point you want to be able to go back to later.

Process

First, remove any process from sheet **Process-Commodity** that consumes or produces a commodity that is no longer mentioned in sheet **Commodity**. For *New Sealand*, this leaves us with three processes: *Gas plant*, *Slack powerplant*, *Wind park*. The output ratio **0.6** of the *Gas plant* is the electric efficiency.

Table 2: Sheet **Process-Commodity**

Process	Commodity	Direction	ratio
Gas plant	CO2	Out	0.2
Gas plant	Elec	Out	**0.6**
Gas plant	Gas	In	1.0
Slack powerplant	CO2	Out	0.0
Slack powerplant	Elec	Out	1.0
Slack powerplant	Slack	In	1.0
Wind park	Elec	Out	1.0
Wind park	Wind	In	1.0

With only these processes remaining, the sheet **Process**, needs some work, too. create an entry for each process that can be built at a given site. The upper capacity limits `cap-up` for each process are the most important figure. *Qlyph Archipelago* is known for its large areas suitable for wind parks up to 200 MW, only surpased by the great offshore sites of *Jepid Island* with 250 MW potential capacity. The other islands only have space for up to 120 MW or 80 MW. *Gas plants* can be built up to 100 MW on every island, except for *Vled Haven*, which can house up to 80 MW only.

Slack powerplants are set to an installed capacity `inst-cap` higher than the peak demand in each site, so that any residual load could always be covered. To make its use unattractive, you set the variable costs `var-cost` to 9 M€/MWh. This yields the following table:

Table 3: Sheet **Process**

Site	Process	inst-cap	cap-lo	cap-up	max-grad	inv-cost	fix-cost	var-cost	wacc	depr.
Jepid Island	Gas plant	25	0	100	5	450000	6000	1.62	0.07	30
Jepid Island	Slack powerplant	999	999	999	inf	0	0	**9000000.0**	0.07	1
Jepid Island	Wind park	0	0	**250**	inf	900000	30000	0.0	0.07	25
Qlyph Archipelago	Gas plant	0	0	100	5	450000	6000	1.62	0.07	30
Qlyph Archipelago	Slack powerplant	999	999	999	inf	0	0	**9000000.0**	0.07	1
Qlyph Archipelago	Wind park	0	0	**200**	inf	900000	30000	0.0	0.07	25
Stryworf Key	Gas plant	25	0	100	5	450000	6000	1.62	0.07	30
Stryworf Key	Slack powerplant	999	999	999	inf	0	0	**9000000.0**	0.07	1
Stryworf Key	Wind park	0	0	**120**	inf	900000	30000	0.0	0.07	25
Vled Haven	Gas plant	0	0	80	5	450000	6000	1.62	0.07	30
Vled Haven	Slack powerplant	999	999	999	inf	0	0	**9000000.0**	0.07	1
Vled Haven	Wind park	0	0	**80**	inf	900000	30000	0.0	0.07	25

Transmission

On transmission, map the network topology of *New Sealand*. *Vled Haven* is the central hub of the state, with the other islands connected like a star shape. The investment costs are scaled according to the air distance from the population centers of each island. So *Jepid Island* with 1.1 M€/MW investment costs is more than twice as far away from *Vled Haven* than *Ylyph Archipelago* with only 0.5 M€/MW. *Stryworf Key* is somewhere between with 0.8 M€/MW. All investment costs are per direction. So the bidirectional cable costs are actually the summed `inv-cost` for both directions.

Table 4: Sheet **Transmission**

Site In	Site Out	Transmission	Commodity	eff	inv-cost	fix-cost	var-cost	inst-cap	cap-lo	cap-up	wacc	depr.
Jepid Island	Vled Haven	undersea	Elec	0.85	**1100000**	30000	0	0	0	inf	0.07	30
Qlyph Archipelago	Vled Haven	undersea	Elec	0.95	**500000**	15000	0	0	0	inf	0.07	30
Stryworf Key	Vled Haven	undersea	Elec	0.9	**800000**	22500	0	0	0	inf	0.07	30
Vled Haven	Jepid Island	undersea	Elec	0.85	1100000	30000	0	0	0	inf	0.07	30
Vled Haven	Qlyph Archipelago	undersea	Elec	0.95	500000	15000	0	0	0	inf	0.07	30
Vled Haven	Stryworf Key	undersea	Elec	0.9	800000	22500	0	0	0	inf	0.07	30

Storage

Storing electricity is possible only on *Qlyph Archipelago*, using an unsepcified technology simply labeled *gravity* here. To allow for parameterising a host of technologies, costs for both storage power and capacity can be specified independently. For most technologies, one of the costs will be dominating, so the other value can be set simply (near) zero to reflect that. The last parameter `init` specifies a) how full the storage is at the first time step and b) at least how full it must be at the final time step. That way, a short simulation duration may not just exhaust the storage.

Table 5: Sheet **Storage** (1/2)

Site	Storage	Commodity	inst-cap-c	cap-lo-c	cap-up-c	inst-cap-p	cap-lo-p	cap-up-p	eff-in	eff-out
Qlyph Archipelago	gravity	Elec	0	0	inf	0	0	inf	0.95	0.95

Table 6: Sheet **Storage** (2/2)

Site	Storage	Commodity	inv-cost-p	inv-cost-c	fix-cost-p	fix-cost-c	var-cost-p	var-cost-c	depr.	wacc	init
Qlyph Archipelago	gravity	Elec	500000	5	0	0.25	0.02	0	50	0.07	0.05

Hacks

In the base scenario, no limit on CO2 emissions from *Gas plants* is needed. Therefore, you set the value to `inf`:

Table 7: Sheet **Hacks**

Name	Value
Global CO2 limit	**inf**

Time series

The only commodity of type `SupIm` is *Wind*, which you defined in sheet **Commodity** on all four islands. Therefore, in total 4 time series must be provided here, even if they are all zeros. As your data provider has not kept his promise to send you the data on time, you (ab)use the `mimo-example.xlsx` data once more, and simply use its time series. To get qualitatively correct results, you assign the best (3600 full load hours) to *Jepid island*, the second best to *Vled Haven* (3000 full load hours) and two copies of the worst time series (2700 full load hours) to *Qlyph Archipelago* and *Stryworf Key*. With that, you get the following table of capacity factors:

Table 8: Sheet **SupIm**

t	Jepid Island.Wind	Qlyph Archipelago.Wind	Stryworf Key.Wind	Vled Haven.Wind
0	0.0	0.0	0.0	0.0
1	0.603	0.935	0.935	0.458
2	0.585	0.942	0.942	0.453
3	0.571	0.956	0.956	0.453
4	0.561	0.956	0.956	0.461
…	…	…	…	…

You make sure that both the island names and the commodity name *exactly* match the identifiers used on the other sheets.

For the demand, you also have no real data for now. But with some scaling (divide by 1000), the example series make for a good temporary demand time series. *Vled Haven* has the highest peak load with 75 MW, followed by *Stryworf Key* with 19 MW and the other islands with 8.2 MW each:

Table 9: Sheet **Demand**

t	Jepid Island.Elec	Qlyph Archipelago.Elec	Stryworf Key.Elec	Vled Haven.Elec
0	0	0	0	0
1	4	4	11	43
2	4	4	10	41
3	4	4	10	40
4	4	4	10	40
…	…	…	…	…

Note: For reference, this is how `newsealand.xlsx` looks for me having performed the above steps.

Test-drive the input

Now that `newsealand.xlsx` is ready to go, start `ipython` in the console. Execute the following lines, best by manually typing them in one by one. *(Hint: use tab completion to avoid typing out function or file names!)*

First, load the data:

```
>>> import urbs
>>> input_file = 'newsealand.xlsx'
>>> data = urbs.read_excel(input_file)
```

`data` now is a standard Python `dict`. So `data.keys()` yields the worksheet names, while `data['commodity']` contains the *Commodity* worksheet as a `DataFrame`. Now create a range:

```
>>> offset, duration = (3500, 14*24)
>>> timesteps = range(offset, offset + duration + 1)

[3500, 3501, ..., 3836]
```

Now you can create the optimisation model, then convert it to an optimisation problem that can be handed to the solver:

```
>>> prob = urbs.create_model(data, timesteps)
```

Now the only thing missing is the solver. It can be used through another object that is generated by the `SolverFactory` function from the `pyomo` package:

```
>>> import pyomo.environ
>>> from pyomo.opt.base import SolverFactory
>>> optim = SolverFactory('glpk')
```

Ignore the deprecation warning[1] for now. The solver object has a `solve` method, which takes the problem as an argument and returns a solution. For bigger problems, the next step can take several hours or even days. Therefore, you enable visible progress output by setting the option `tee`[2]. Additionally, you can save the output to a logfile using the `logfile` option:

[1] If you used Coopr 4.0, simply `import coopr.environ` before importing `SolverFactory`.

[2] like the GNU tee output redirection tool.

```
>>> result = optim.solve(prob, logfile='solver.log', tee=True)
```

This results in roughly the following output appearing on the console:

```
GLPSOL: GLPK LP/MIP Solver, v4.55
[...]
GLPK Simplex Optimizer, v4.55
26275 rows, 22558 columns, 63630 non-zeros
Preprocessing...
14793 rows, 13120 columns, 35970 non-zeros
Scaling...
 A: min|aij| = 2.305e-003  max|aij| = 1.053e+000  ratio = 4.567e+002
GM: min|aij| = 3.606e-001  max|aij| = 2.773e+000  ratio = 7.691e+000
EQ: min|aij| = 1.300e-001  max|aij| = 1.000e+000  ratio = 7.691e+000
Constructing initial basis...
Size of triangular part is 14790
      0: obj =   3.000000000e+005  infeas = 2.158e+004 (3)
    500: obj =   2.443067336e+007  infeas = 8.024e+003 (3)
   1000: obj =   3.635166806e+011  infeas = 5.311e+003 (3)
*  1379: obj =   1.688377193e+012  infeas = 0.000e+000 (3)
[...]
*  5500: obj =   3.438413434e+007  infeas = 6.221e-014 (3)
*  5822: obj =   3.419699391e+007  infeas = 7.889e-031 (3)
OPTIMAL LP SOLUTION FOUND
Time used:   3.5 secs
Memory used: 25.3 Mb (26496968 bytes)
Writing basic solution to '<temporary.glpk.raw>'...
48835 lines were written
```

Finally, you can load the result back into the optimisation problem oject `prob`:

```
>>> prob.solutions.load_from(result)
True
```

This object now contains all input data, the equations and result data. If you store this object as a file, you can later always create new analyses from it. That's what `save()` is made for:

```
>>> urbs.save(prob, 'newsealand-base.pgz')
```

This becomes especially helpful for large problems that take hours to solve. Back to the `prob`. To get a quick numerical overview on the most important result numbers, use `report()`:

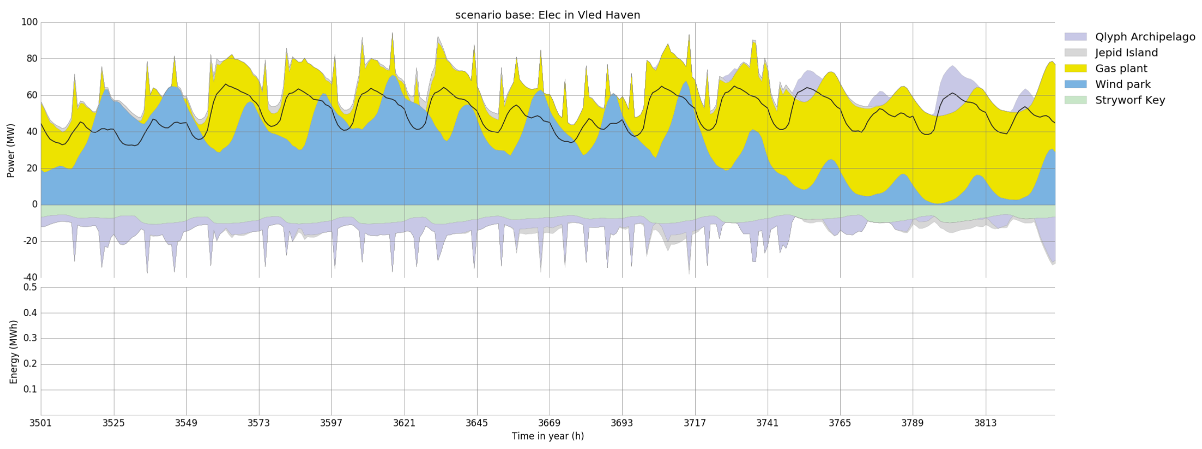
```
>>> urbs.report(prob, 'report.xlsx',prob.com_demand,prob.sit)
```

By default, this report only includes total costs and capacities of process, transmission and storage. By adding the optional third and fourth parameter, you can retreive timeseries listings of energy production per site. For now, you are only interested in *electricity* in *Vled Haven*:

```
>>> urbs.report(prob, 'report-vled-haven.xlsx',
...             ['Elec'], ['Vled Haven'])
```

Then you finally want to see how the electricity production *looks* like. For that you use `plot()`:

```
>>> %matplotlib
>>> fig = urbs.plot(prob, 'Elec', 'Vled Haven')
```



Depending on the plotting backend, you now either see a window with the plot ('TkAgg', 'QtAgg'), or nothing. Either way, you can save the figure to a file using:

```
>>> fig.savefig('newsealand-base-elec-vled-haven.png',
...             dpi=300, bbox_inches='tight')
```

The file extension determines how the output is written. Among the supported formats are jpg, pdf, png, svg and tif. Use `png` if raster images are needed and rely on `pdf` or `svg` for vector output. The `dpi` option is only used for raster images. `bbox_inches='tight'` removes unnecessary whitespace around the plot, making it suitable for inclusion in reports or presentations.

Create a run script

As it is quite tedious to perform the above actions by hand all the time, a script can automate these. This is where a `runme.py` script becomes handy.

Create a copy of the script file `runme.py` and give it a suitable name, e.g. `runns.py`.

Modify the `scenario_co2_limit` function. As the base scenario now has no limit, reducing it by 95 % does not make it finite. Therefore you set a fixed hard (annual) limit of 40 million tonnes of CO2 equivalent:

```
def scenario_co2_limit(data):
    # change global CO2 limit
    hacks = data['hacks']
    hacks.loc['Global CO2 limit', 'Value'] = 40000
    return data
```

Next, set adjust the plot_tuples and report_tuples by replacing `North`, `Mid` and `South` by the four islands of Newsealand. Furthermore, you want to show imported/exported electricity in the plots in custom colors. So you modify the `my_colors dict` like this:

```
my_colors = {
    'Vled Haven': (230, 200, 200),
    'Stryworf Key': (200, 230, 200),
    'Qlyph Archipelago': (200, 200, 230),
    'Jepid Island': (215,215,215)}
```

Finally, you head down to the `if __name__ == '__main__'` section that is executed when the script is called. There, you change the `input_file` to your spreadsheet `newsealand.xlsx` and increase the optimisation duration to 14 days (`14*24` time steps). For now, you don't need the other scenarios, so you exclude them from the `scenarios list`:

```
if __name__ == '__main__':
    input_file = 'newsealand.xlsx'
    result_name = os.path.splitext(input_file)[0]  # cut away file␣
↪extension
    result_dir = prepare_result_directory(result_name)  # name + time␣
↪stamp

    (offset, length) = (3500, 14*24)  # time step selection
    timesteps = range(offset, offset+length+1)

    # select scenarios to be run
    scenarios = [
        scenario_base,
        scenario_co2_limit]

    for scenario in scenarios:
        prob = run_scenario(input_file, timesteps, scenario, result_dir)
```

Note: For reference, here is how `runns.py` looks for me.

1.2 Technical documentation

Continue here if you want to automate the scripting further, understand the model equations or extend the model yourself.

1.2.1 Reporting function explained

This page is a "code walkthrough" through the function `report()`. It shows more technical details than the *Tutorial* or *Workflow* pages, to facilitate writing one's own analysis scripts that directly retrieve variables from the optimisation:

Report

So let's start by first printing the function as a whole:

After the function header and the docstring briefly explaining its use, another function, `get_constants()`, is called. Before really diving into the report function, first one of the two *Retrieve results* functions is presented.

Get constants

Taking only one argument, this function retrieves all time-independent quantities from the given optimisation problem object and returns them as a `tuple()` of `DataFrame`. The low-level access functions `get_entity()` and `get_entities()` are beyond the scope of this walk through. They both yield "raw" `DataFrame` objects with only minor pre-processing of index names.

The second paragraph deals with the emission timeseries `co2` by calculating its sum over time. The `unstack()` method allows to move the time dimension (index level `0` or the first) into the column direction. To sum over time, method `sum()` is called with its `axis` argument set to columns (`1`). This yields a DataFrame indexed over the tuple *(site, process, input commodity, output commodity)* and the summed emissions as value.

Get timeseries

With the arguments `instance`, `com` and `sit` the function :func: *get_timeseries* returns `DataFrames` of all timeseries that are referring to the given commodity and site. This includes the derivative for `created` and `consumed`, which is calculated and standardized by the power capacity at the end of the function.

Write to Excel

The ExcelWriter class creates a writer object, which is then used by the `to_excel()` method calls to aggregate all outputs into a single spreadsheet.

Note: `to_excel()` can also be called with a filename. However, this overwrites an existing file completely, thus deleting existing sheets. For quickly saving a `DataFrame`, to a spreadsheet, a simple `df.to_excel('df.xlsx', 'df')` is sufficient.

Constants

As written already, the individual `DataFrame` objects are written to individual sheets within the same spreadsheet file by using the writer object as a target. `co2` is an exception, as it starts as a `Series`. It must be first converted to a DataFrame by calling `to_frame()`.

Timeseries

Initialize an empty `list()` and an empty `dict()` for collecting the timeseries data. These are two builtin Python data structures. `energies` will become a list of `DataFrame` objects before getting stitched together, while `timeseries` becomes a dictionary of `DataFrame` objects, with a tuple `(commodity, site)` as key.

Module function *`get_timeseries()`* is similar to *`get_constants()`*, just for time-dependent quantities. For a given commodity and site, this function returns all DataFrames needed to create a balance plot.

Only overproduction is calculated in place. While it should not happen for scenarios close to today's situation, future scenarios with much excess renewable infeed, overproduction could happen for significant duration and amount.

Using the function `pandas.concat()`, multiple DataFrames are glued together next to each other (`axis=1`), while creating a nested column index wih custom labels (`keys=...`) for each of the list argument (`[...]`). The resulting timeseries tableau is copied to the corresponding place in the `timeseries` dictionary.

For the *Energy sums* sheet, all timeseries DataFrames are summed along the time axis, resulting in a Series for each timeseries. These are then glued together on top of each other (`axis=0`) with a nested row index with custom labels (`keys=...`) for each series type. Finally the Series is converted back to a DataFrame, using `Commodity.Site` as the column title template.

Finally, the *Energy sums* table is assembled by stitching together the individual energy sums per commodity and site and filling missing values with `fillna()`.

Finally, the *timeseries* tables are saved without change to individual sheets.

1.2.2 urbs.py module description

Overview

The following is a minimum "hello world" script that shows the life cycle of the optimization object *prob*, and how the various `urbs` module functions create it, modify it and process it.:

```
import urbs
from pyomo.opt.base import SolverFactory

# read input, create optimisation problem
data = urbs.read_excel('mimo-example.xlsx')
prob = urbs.create_model(data)

# solve problem, read results
optim = SolverFactory('glpk')
result = optim.solve(prob)
prob.solutions.load_from(result)

# save problem instance (incl. input and result) for later analyses
urbs.save(prob, 'mimo-example.pgz')

# write report and plot timeseries
urbs.report(prob, 'report.xlsx')
urbs.plot(prob, 'Elec', 'Mid')
```

The following lists and describes the use of all module-level functions. They are roughly ordered from high-level to low-level access, followed by helper functions.

Create model

`urbs.`**`read_excel`**(*filename*)

Parameters **`filename`** (`str`) – spreadsheet filename

Returns urbs input dict

The spreadsheet must contain 7 sheets labelled 'Commodity', 'Process', 'Process-Commodity', 'Transmission', 'Storage', 'Demand' and 'SupIm'. It can contain 2 additional sheets called 'Buy-Sell-Price' and 'Hacks'. If present, function `add_hacks()` is called by `create_model()` upon model creation.

Refer to the *mimo-example.xlsx* file for exemplary documentation of the table contents and definitions of all attributes by selecting the column titles.

`urbs.`**`create_model`**`(`*data*, *timesteps*`)`

Returns a Pyomo *ConcreteModel* object.

Parameters

- **`data`** (*`dict`*) – input like created by *`read_excel()`*
- **`timesteps`** (*`list`*) – consecutive list of modelled timesteps

Returns urbs model object

Timestep numbers must match those of the demand and supim timeseries.

If argument `data` has the key `'hacks'`, function *`add_hacks()`* is called with `data['hacks']` as the second argument.

`urbs.`**`add_hacks`**`(`*model*, *hacks*`)`

Is called by *`create_model()`* to add special elements, e.g. constraints, to the model. Each hack, if present, can trigger the creation of additional sets, parameters, variables or constraints. Refer to the code of this function to see which hacks exists and what they do.

As of v0.3, only one hack exists: if a line "Global CO2 limit" exists in the hacks DataFrame, its value is used as a global upper limit for a constraint that limits the annual creation of the commodity "CO2".

param model urbs model object (not instance!)

param hacks a DataFrame of hacks

return model the modified urbs model object

Report & plotting

These two **high-level** functions cover the envisioned use of the unmodified urbs model and should cover most use cases.

`urbs.`**`plot`**`(`*prob*, *com*, *sit*[, *timesteps=None*]`)`

Parameters

- **`prob`** – urbs model instance
- **`com`** (*`str`*) – commodity name to plot
- **`sit`** (*`str`*) – site name to plot
- **`timesteps`** (*`list`*) – timesteps to plot, default: all

Return fig matplotlib figure handle

`urbs.`**`report`**`(`*prob*, *filename*, *commodities*, *sites*`)`

Write optimisation result summary to spreadsheet.

Parameters

- **`prob`** – urbs model instance
- **`filename`** (*`str`*) – spreadsheet filename, will be overwritten if exists
- **`commodities`** (*`list`*) – list of commodities for which to output timeseries
- **`sites`** (*`list`*) – list sites for which to output timeseries

Retrieve results

While `report()` and `plot()` are quite flexible, custom result analysis scripts might be needed. These can be built on top of the following two **medium-level** functions. They retrieve all time-dependent and -independent quantities and return them as ready-to-use DataFrames.

`urbs.get_constants`(*prob*)
Return summary DataFrames for time-independent variables

Parameters **prob** – urbs model instance

Returns tuple of constants (costs, process, transmission, storage)

`urbs.get_timeseries`(*prob*, *com*, *sit*, *timesteps=None*)
Return DataFrames of all timeseries referring to a given commodity and site

Parameters

- **prob** – urbs model instance
- **com** (`str`) – commodity name
- **sit** (`str`) – site name
- **timesteps** (`list`) – timesteps, default: all modelled timesteps

Returns

tuple of timeseries (created, consumed, storage, imported, exported) tuple of DataFrames timeseries. These are:

- created: timeseries of commodity creation, including stock source
- consumed: timeseries of commodity consumption, including demand
- storage: timeseries of commodity storage (level, stored, retrieved)
- imported: timeseries of commodity import (by site)
- exported: timeseries of commodity export (by site)

Persistence

To store valuable results for later analysis, or cross-scenario comparisons weeks after the original run, saving a problem instance with loaded results makes it possible to use one's comparison scripts without having to solve the optimisation problem again. Simply `load()` the previously stored object using `save()`:

`urbs.save`(*prob*, *filename*)
Save rivus model instance to a gzip'ed pickle file

Pickle is the standard Python way of serializing and de-serializing Python objects. By using it, saving any object, in case of this function a Pyomo ConcreteModel, becomes a twoliner.

GZip is a standard Python compression library that is used to transparently compress the pickle file further.

It is used over the possibly more compact bzip2 compression due to the lower runtime. Source: <http://stackoverflow.com/a/18475192/2375855>

Parameters

- **prob** – a rivus model instance
- **filename** (*str*) – pickle file to be written

Returns nothing

`urbs.load(filename)`

Load a rivus model instance from a gzip'ed pickle file

Parameters **filename** (*str*) – pickle file

Return prob the unpickled rivus model instance

Low-level access

If the previous functions still don't cut it, there are three **low-level** functions.

`urbs.list_entities(prob, entity_type)`

Parameters

- **prob** – urbs model instance
- **entity_type** (*str*) – allowed values: set, par, var, con, obj

Returns a DataFrame with name, description and domain of entities

`urbs.get_entity(prob, name)`

Parameters

- **prob** – urbs model instance
- **name** (*str*) – name of a model entity

Returns Series with values of model entity

`urbs.get_entities(prob, names)`

Parameters

- **prob** – urbs model instance
- **name** (*list*) – list of model entity names

Returns DataFrame with values entities in columns

Only call `get_entities` for entities that share identical domains. This can be checked with *`list_entities()`*. For example, variable `cap_pro` naturally has the same domain as `cap_pro_new`.

Helper functions

`urbs.annuity_factor(n, i)`

Annuity factor formula.

Evaluates the annuity factor formula for depreciation duration and interest rate. Works also well for equally sized numpy arrays as input.

Parameters

- **n** (*int*) – number of depreciation periods (years)

- **i** (*float*) – interest rate (e.g. 0.06 means 6 %)

Returns value of the expression $\frac{(1+i)^n i}{(1+i)^n - 1}$

`urbs.`**`commodity_balance`**(*m*, *tm*, *sit*, *com*)

Calculate commodity balance at given timestep.

For a given commodity, site and timestep, calculate the balance of consumed (to process/storage/transmission, counts positive) and provided (from process/storage/transmission, counts negative) energy. Used as helper function in `create_model()` for defining constraints on demand and stock commodities.

Parameters

- **m** – the ConcreteModel object
- **tm** – the timestep number
- **sit** – the site
- **co** – the commodity

Returns amount of consumed (positive) or provided (negative) energy

`urbs.`**`split_columns`**(*columns*[, *sep='.'*])

Given a list of column labels containing a separator string (default: '.'), derive a MulitIndex that is split at the separator string.

Parameters

- **columns** (*list*) – column labels, each containing the separator string
- **sep** (*str*) – the separator string (default: '.')

Returns a MultiIndex corresponding to input, with levels split at separator

`urbs.`**`to_color`**(*obj=None*)

Assign a deterministic pseudo-random color to argument.

If `COLORS[obj]` is set, return that. Otherwise, create a deterministically random color from the `hash()` of that object. For strings, this value depends only on the string content, so that identical strings always yield the same color.

Parameters **obj** – any hashable object

Returns a *(r,g,b)* tuple if COLORS[obj] exists, otherwise a hexstring

`urbs.`**`COLORS`**

`dict` of process and site colors. Colors are stored as *(r,g,b)* tuples in range *0-255* each. To retrieve a color in a form usable with matplotlib, used the helper function `to_color()`.

This snippet from the example script *runme.py* shows how to add custom colors:

```
# add or change plot colours
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
```

1.2.3 Mathematical Documentation

In this Section, **mathematical description** of the model will be explained. This includes listing and describing all relevant sets, parameters, variables and constraints using mathematical notation together with 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 10: *Table: Model Sets*

Set	Description
$t \in T$	Timesteps
$t \in T_\text{m}$	Modelled Timesteps
$v \in V$	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** $T = \{t_0, \dots, t_N\}$ 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 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 later required to facilitate the definition of the storage state equation. In script `urbs.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

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 `urbs.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 `urbs.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 `SupIm`, `Stock`, `Demand`, `Env`, `Buy`, `Sell`. In script `urbs.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 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 `urbs.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 `urbs.py` this set is defined as `sto` and initalized 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 `urbs.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 : `Investment`, `Fix`, `Variable`, `Fuel`, `Revenue`, `Purchase`, `Startup` . In script `urbs.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. As a result a collection of by the same rule defined tuples, represents a tuple set.

Commodity Tuples

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

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

Process Tuples

Process tuples represent combinations of possible processes. These are represented by the set P_v. A member p_v in set P_v is a process p in site v. For example, *(North, Coal Plant)* is interpreted as process *Coal Plant* in site *North*. This set is defined as `pro_tuples` and given by the code fragment:

```
m.pro_tuples = pyomo.Set(
    within=m.sit*m.pro,
```

(continues on next page)

(continued from previous page)

```
    initialize=m.process.index,
    doc='Combinations of possible processes, e.g. (North,Coal plant)')
```

A subset of these process tuples `pro_partial_tuples` P_v^partial is formed in order to identify processes that have partial & startup properties. Programmatically, they are identified by those processes, which have the parameter `ratio-min` set for one of their input commodities in table *Process-Commodity*. The tuple set is defined as:

```
m.pro_partial_tuples = pyomo.Set(
    within=m.sit*m.pro,
    initialize=[(site, process)
                for (site, process) in m.pro_tuples
                for (pro, _) in m.r_in_min_fraction.index
                if process == pro],
    doc='Processes with partial input')
```

A second subset is formed in order to caputure 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.sit*m.pro,
    initialize=m.proc_area.index,
    doc='Processes and Sites with area Restriction')
```

Transmission Tuples

Transmission tuples represent combinations of possible transmissions. These are represented by the set $F_{cv_\text{out}v_\text{in}}$. A member $f_{cv_\text{out}v_\text{in}}$ in set $F_{cv_\text{out}v_\text{in}}$ is a transmission f,that is directed from an origin site v_out to a destination site v_{in} and carries a commodity c. 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, *(South, Mid, hvac, Elec)* is interpreted as transmission *hvac* that is directed from origin site *South* to destination site *Mid* carrying commodity *Elec*. This set is defined as `tra_tuples` and given by the code fragment:

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

Additionally, Subsets F_{vc}^exp and F_{vc}^imp represents all exporting and importing transmissions of a commodity c in a site v. These subsets can be obtained by fixing either the origin site(for export) v_out or the destination site(for import) v_in to a desired site v in tuple set $F_{cv_\text{out}v_\text{in}}$.

Storage Tuples

Storage tuples represent combinations of possible storages by site. These are represented by the set S_{vc}. A member s_{vc} in set S_{vc} is a storage s of commodity c in site v For example, *(Mid, Bat, Elec)* is interpreted as storage *Bat* of commodity *Elec* in site *Mid*. This set is defined as `sto_tuples` and given by the code fragment:

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

Process Input Tuples

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

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

Where: `r_in` represents the process input ratio.

For processes in the tuple set `pro_partial_tuples` $C_{vp}^{\text{in,partial}}$, the following tuple set `pro_partial_input_tuples` enumerates their input commodities. It is used to index the constraints that determine a process' input commodity flow (i.e. `def_process_input` and `def_partial_process_input`). It is defined by the following code fragment:

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

Process Output Tuples

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

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

Where: `r_out` represents the process output ratio.

Demand Side Management Tuples

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

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

The second kind $D_{vct,tt}^{\text{down}}$ 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 site v and commodity c the tuples contain two time indices. For example *(5001, 5003, Mid, Elec)* is intepreted as the downshift in timestep *5003*, which was caused by the upshift of timestep *5001* in 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.sit*m.com,
    initialize=[(t, tt, site, commodity)
                for (t,tt, 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,
↪Mid,Elec)')
```

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.

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 seperated into sections by their area of use. These Sections are Cost, Commodity, Process, Transmission and Storage.

Table 11: *Table: Model Variables*

Variable	Unit	Description
Cost Variables		
ζ	€/a	Total System Cost
ζ_{inv}	€/a	Investment Costs
ζ_{fix}	€/a	Fix Costs
ζ_{var}	€/a	Variable Costs
ζ_{fuel}	€/a	Fuel Costs
ζ_{rev}	€/a	Revenue Costs
ζ_{pur}	€/a	Purchase Costs
ζ_{startup}	€/a	Startup Costs
Commodity Variables		
ρ_{vct}	MW	Stock Commodity Source Term
ϱ_{vct}	MW	Sell Commodity Source Term
ψ_{vct}	MW	Buy Commodity Source Term
Process Variables		
κ_{vp}	MW	Total Process Capacity
$\hat{\kappa}_{vp}$	MW	New Process Capacity
τ_{vpt}	MW	Process Throughput
$\epsilon_{vcpt}^{\text{in}}$	MW	Process Input Commodity Flow
$\epsilon_{vcpt}^{\text{out}}$	MW	Process Output Commodity Flow
ω_{vpt}	MW	Process Online Capacity
ϕ_{vpt}	MW	Process Startup Capacity
Transmission Variables		
κ_{af}	MW	Total transmission Capacity
$\hat{\kappa}_{af}$	MW	New Transmission Capacity
π_{aft}^{in}	MW	Transmission Power Flow (Input)
π_{aft}^{out}	MW	Transmission Power Flow (Output)
Storage Variables		
κ_{vs}^{c}	MWh	Total Storage Size
$\hat{\kappa}_{vs}^{\text{c}}$	MWh	New Storage Size
κ_{vs}^{p}	MW	Total Storage Power
$\hat{\kappa}_{vs}^{\text{p}}$	MW	New Storage Power
$\epsilon_{vst}^{\text{in}}$	MW	Storage Power Flow (Input)
$\epsilon_{vst}^{\text{out}}$	MW	Storage Power Flow (Output)
$\epsilon_{vst}^{\text{con}}$	MWh	Storage Energy Content
Demand Side Management Variables		
δ_{vct}^{up}	MW	DSM Upshift
$\delta_{vct,tt}^{\text{down}}$	MW	DSM Downshift

Cost Variables

Total System Cost, ζ : the variable ζ represents the *annual total expense incurred* in reaching the satisfaction of the given energy demand. This 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,
```

(continues on next page)

(continued from previous page)

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

More information on calculation of this variable is available at the section *Objective function*.

Total System costs by type: System costs are divided into 6 cost types by their meaning and purpose. The separation of costs by type, facilitates business planning and provides calculation accuracy As mentioned before these cost types are hardcoded, which means they are not considered to be fixed or changed by the user. These cost types are as following;

Investment Costs ζ_{inv}: The variable ζ_{inv} represents the annualised total investment costs. Costs for required new investments on storage, process and transmission technologies.

Fix Costs ζ_{fix}: The variable ζ_{fix} represents the annualised total fix costs. Fix costs for all used storage, process, and transmission technologies. Such as maintenance costs.

Variable Costs ζ_{var}: The variable ζ_{var} represents the annualised total variables costs. Variable costs that are reliant on the usage amount and period of the storage, process, transmission technologies.

Fuel Costs ζ_{fuel}: The variable ζ_{fuel} represents the annualised total fuel costs. Fuel costs are dependent on the usage of stock commodities($\forall c \in C_{\text{stock}}$).

Revenue Costs ζ_{rev}: The variable ζ_{rev} represents the annualised total revenue costs. Revenue costs is defined for the costs that occures by selling the sell commodities($\forall c \in C_{\text{sell}}$). Since this variable is an income for the system, it is either zero or has a negative value.

Purchase Costs ζ_{pur}: The variable ζ_{pur} represents the annualised total purchase costs. Purchase costs is defined for the costs that occures by buying the buy commodities ($\forall c \in C_{\text{buy}}$).

Startup Costs ζ_{startup}: The variable ζ_{startup} represents the annualised total startup costs. Startup costs are reliant on the yearly startup occurences of the processes.

For more information on calculation of these variables see section *Objective function*.

Commodity Variables

Stock Commodity Source Term, ρ_{vct}, `e_co_stock`, MW : The variable ρ_{vct} represents the energy amount in [MW] that is being used by the system of commodity c from type stock ($\forall c \in C_{\text{stock}}$) in a site v ($\forall v \in V$) at timestep t ($\forall t \in T_{\text{m}}$). In script `urbs.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 (MW) per timestep')
```

Sell Commodity Source Term, ϱ_{vct}, `e_co_sell`, MW : The variable ϱ_{vct} represents the energy amount in [MW] that is being used by the system of commodity c from type sell ($\forall c \in C_{\text{sell}}$) in a site v ($\forall v \in V$) at timestep t ($\forall t \in T_{\text{m}}$). In script `urbs.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 (MW) per timestep')
```

Buy Commodity Source Term, ψ_{vct}, `e_co_buy`, MW : The variable ψ_{vct} represents the energy amount in [MW] that is being used by the system of commodity c from type buy ($\forall c \in C_{\text{buy}}$) in a site v ($\forall v \in V$) at timestep t ($\forall t \in T_{\text{m}}$). In script `urbs.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 (MW) per timestep')
```

Process Variables

Total Process Capacity, κ_{vp}, `cap_pro`: The variable κ_{vp} represents the total potential throughput (capacity) of a process tuple p_v ($\forall p \in P, \forall v \in V$), 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 and helps us to calculate a more accurate cost plan. For further information see Process Capacity Rule. This variable is expressed in the unit MW. In script `urbs.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, $\hat{\kappa}_{vp}$, `cap_pro_new`: The variable $\hat{\kappa}_{vp}$ represents the capacity of a process tuple p_v ($\forall p \in P, \forall v \in V$) that needs to be installed additionally to the energy system in order to provide the optimal solution. This variable is expressed in the unit MW. In script `urbs.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)')
```

Process Throughput, τ_{vpt}, `tau_pro` : The variable τ_{vpt} represents the measure of (energetic) activity of a process tuple p_v ($\forall p \in P, \forall v \in V$) at a timestep t ($\forall t \in T_m$). By default, process throughput is represented by the major input commodity flow of the process (e.g. 'Gas' for 'Gas plant', 'Wind' for 'Wind park'). 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**. This variable is expressed in the unit MW. In script `urbs.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 (MW) through process')
```

Process Input Commodity Flow, $\epsilon^{\text{in}}_{vcpt}$, `e_pro_in`: The variable $\epsilon^{\text{in}}_{vcpt}$ represents the flow input into a process tuple p_v ($\forall p \in P, \forall v \in V$) 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 MW. In script `urbs.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 per timestep')
```

Process Output Commodity Flow, $\epsilon_{vcpt}^{\text{out}}$, `e_pro_out`: The variable $\epsilon_{vcpt}^{\text{out}}$ represents the flow output out of a process tuple p_v ($\forall p \in P, \forall v \in V$) 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 MW (or tonnes e.g. for the environmental commodity 'CO2'). In script `urbs.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 per timestep')
```

Process Online Capacity, ω_{vpt}, `cap_online`: This variable is the time-dependent version of the usual process capacity κ_{vp}. It is defined for partial process tuples, i.e. those processes that have the parameter input ratio `ratio-min` set. of a process tuple p_v ($\forall p \in P, \forall v \in V$) at a timestep t ($\forall t \in T$). In script `urbs.py` this variable is defined by the model variable `onlinestatus` and initialized by the following code fragment:

```
m.cap_online = pyomo.Var(
    m.t, m.pro_partial_tuples,
    within=pyomo.NonNegativeReals,
    doc='Online capacity (MW) of process per timestep')
```

Process Startup Capacity, ϕ_{vpt}', `startup_pro`: This variable indicates every rise in the *process online capacity*. This indicator is then used to determine startup costs for all partial process tuples. The variable is defined by the following code fragment:

```
m.startup_pro = pyomo.Var(
    m.tm, m.pro_partial_tuples,
    within=pyomo.NonNegativeReals,
    doc='Started capacity (MW) of process per timestep')
```

Transmission Variables

Total Transmission Capacity, κ_{af}, `cap_tra`: The variable κ_{af} represents the total potential transfer power of a transmission tuple f_{ca}, 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 `urbs.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,
    within=pyomo.NonNegativeReals,
    doc='Total transmission capacity (MW)')
```

New Transmission Capacity, $\hat{\kappa}_{af}$, `cap_tra_new`: The variable $\hat{\kappa}_{af}$ represents the additional capacity, that needs to be installed, of a transmission tuple f_{ca}, 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 `urbs.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)')
```

Transmission Power Flow (Input), π^{in}_{aft}, `e_tra_in`: The variable π^{in}_{aft} represents the power flow input into 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 MW. 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='Power flow into transmission line (MW) per timestep')
```

Transmission Power Flow (Output), π^{out}_{aft}, `e_tra_out`: The variable π^{out}_{aft} represents the power 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 MW. 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 (MW) per timestep')
```

Storage Variables

Total Storage Size, κ^{c}_{vs}, `cap_sto_c`: The variable κ^{c}_{vs} represents the total load capacity of a storage tuple s_{vc}. 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 `urbs.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, $\hat{\kappa}^{\text{c}}_{vs}$, `cap_sto_c_new`: The variable $\hat{\kappa}^{\text{c}}_{vs}$ 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 `urbs.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)')
```

Total Storage Power, κ_{vs}^{p}, `cap_sto_p`: The variable κ_{vs}^{p} 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 `urbs.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, $\hat{\kappa}_{vs}^{\text{p}}$, `cap_sto_p_new`: The variable $\hat{\kappa}_{vs}^{\text{p}}$ 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 `urbs.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)')
```

Storage Power Flow (Input), $\epsilon_{vst}^{\text{in}}$, `e_sto_in`: The variable $\epsilon_{vst}^{\text{in}}$ represents the input power flow into a storage tuple s_{vc} at a timestep t. Input power flow into a storage tuple can also be defined as the charge of a storage tuple. This variable is expressed in the unit MW. In script `urbs.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='Power flow into storage (MW) per timestep')
```

Storage Power Flow (Output), $\epsilon_{vst}^{\text{out}}$, `e_sto_out`: The variable $\epsilon_{vst}^{\text{out}}$ represents the output power flow out of a storage tuple s_{vc} at a timestep t. Output power flow out of a storage tuple can also be defined as the discharge of a storage tuple. This variable is expressed in the unit MW. 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_out = pyomo.Var(
    m.tm, m.sto_tuples,
    within=pyomo.NonNegativeReals,
    doc='Power flow out of storage (MW) per timestep')
```

Storage Energy Content, $\epsilon_{vst}^{\text{con}}$, `e_sto_con`: The variable $\epsilon_{vst}^{\text{con}}$ 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) in timestep')
```

Demand Side Management Variables

DSM Upshift, δ_{vct}^{up}, `dsm_up`, MW: The variable δ_{vct}^{up} represents the DSM upshift in time step t 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')
```

DSM Downshift, $\delta_{vct,tt}^{\text{down}}$, `dsm_down`, MW: The variable $\delta_{vct,tt}^{\text{down}}$ represents the DSM downshift in timestepp tt caused by the upshift in time t in site v for commodity c. The special combinations of timesteps t and tt for each site and 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')
```

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 data, 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 seperated into two main parts, these are Technical and Economical Parameters.

Technical Parameters

Table 12: *Table: Technical Model Parameters*

Parameter	Unit	Description
General Technical Parameters		
w	_	Weight
Δt	h	Timestep Duration
Commodity Technical Parameters		
d_{vct}	MW	Demand for Commodity
s_{vct}	_	Intermittent Supply Capacity Factor
$\overline{l}_{vc}$	MW	Maximum Stock Supply Limit Per Time Step
$\overline{L}_{vc}$	MW	Maximum Annual Stock Supply Limit Per Vertex
$\overline{m}_{vc}$	t	Maximum Environmental Output Per Time Step
$\overline{M}_{vc}$	t	Maximum Annual Environmental Output
$\overline{g}_{vc}$	MW	Maximum Sell Limit Per Time Step
$\overline{G}_{vc}$	MW	Maximum Annual Sell Limit
$\overline{b}_{vc}$	MW	Maximum Buy Limit Per Time Step
$\overline{B}_{vc}$	MW	Maximum Annual Buy Limit
$\overline{L}_{CO_2}$	t	Maximum Global Annual CO2 Emission Limit

Continued on next page

Table 12 – continued from previous page

Parameter	Unit	Description
Process Technical Parameters		
$\underline{K}_{vp}$	MW	Process Capacity Lower Bound
K_{vp}	MW	Process Capacity Installed
$\overline{K}_{vp}$	MW	Process Capacity Upper Bound
$\overline{PG}_{vp}$	1/h	Process Maximal Power Gradient (relative)
$\underline{P}_{vp}$	_	Process Minimum Part Load Fraction
r_{pc}^{in}	_	Process Input Ratio
$\underline{r}_{pc}^{\text{in}}$	_	Process Partial Input Ratio
r_{pc}^{out}	_	Process Output Ratio
Storage Technical Parameters		
I_{vs}	1	Initial and Final State of Charge
e_{vs}^{in}	_	Storage Efficiency During Charge
e_{vs}^{out}	_	Storage Efficiency During Discharge
$\underline{K}_{vs}^{\text{c}}$	MWh	Storage Content Lower Bound
K_{vs}^{c}	MWh	Storage Content Installed
$\overline{K}_{vs}^{\text{c}}$	MWh	Storage Content Upper Bound
$\underline{K}_{vs}^{\text{p}}$	MW	Storage Power Lower Bound
K_{vs}^{p}	MW	Storage Power Installed
$\overline{K}_{vs}^{\text{p}}$	MW	Storage Power Upper Bound
Transmission Technical Parameters		
e_{af}	_	Transmission Efficiency
$\underline{K}_{af}$	MW	Tranmission Capacity Lower Bound
K_{af}	MW	Tranmission Capacity Installed
$\overline{K}_{af}$	MW	Tranmission Capacity Upper Bound
Demand Side Management Parameters		
e_{vc}	_	DSM Efficiency
y_{vc}	_	DSM Delay Time
o_{vc}	_	DSM Recovery Time
$\overline{K}_{vc}^{\text{up}}$	MW	DSM Maximal Upshift Capacity
$\overline{K}_{vc}^{\text{down}}$	MW	DSM Maximal Downshift Capacity

General Technical Parameters

Weight, w, `weight`: The variable 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 variable represents the rate of a year (8760 hours) to 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 `urbs.py` this variable is defined by the model variable `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 variable 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 variable is the unit of time for the optimization model This variable is expressed in the unit h and by default the value is set to `1`. In script `urbs.py` this variable is defined by the model variable `dt` and initialized by the following code fragment:

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

Commodity Technical Parameters

Demand for Commodity, d_{vct}, `m.demand.loc[tm][sit, com]`: The parameter represents the energy amount of a demand commodity tuple c_{vq} required at a timestep t $(\forall v \in V, q = "Demand", \forall t \in T_m)$. The unit of this parameter is MW. This data is to be provided by the user and to be entered in the spreadsheet. The related section for this parameter in the spreadsheet can be found under the "Demand" 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 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_{vct}, `m.supim.loc[tm][sit, com]`: The parameter s_{vct} represents the normalized availability of a supply intermittent commodity c $(\forall c \in C_{\text{sup}})$ in a 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. 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 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 *SupIm* in any case.

Maximum Stock Supply Limit Per Time Step, $\overline{l}_{vc}$, `m.commodity.loc[sit, com, com_type]['maxperstep']`: The parameter $\overline{l}_{vc}$ represents the maximum energy amount of a stock commodity tuple c_{vq} $(\forall v \in V, q = "Stock")$ that energy model is allowed to use per time step. 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 can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{vq} and the sixth column of stock commodity tuples in this sheet with the header label "maxperstep" represents the parameter $\overline{l}_{vc}$. 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, $\overline{L}_{vc}$, `m.commodity.loc[sit, com, com_type]['max']`: The parameter $\overline{L}_{vc}$ represents the maximum energy amount of a stock commodity tuple c_{vq} $(\forall v \in V, q = "Stock")$ that energy model is allowed to use annually. The unit of this parameter is MW. This parameter is to be provided by the user and to be entered in spreadsheet. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{vq} and the fifth column of stock commodity tuples in this sheet with the header label "max" represents the parameter $\overline{L}_{vc}$. 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 Time Step, $\overline{m}_{vc}$, `m.commodity.loc[sit, com, com_type]['maxperstep']`: The parameter $\overline{m}_{vc}$ represents the maximum energy amount of an environmental commodity tuple c_{vq} $(\forall v \in V, q = "Env")$ 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. This parameter is to be provided by the user and to be entered in spreadsheet. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{vq} and the sixth column of enviromental commodity tuples in this sheet with the header label "maxperstep" represents the parameter $\overline{m}_{vc}$. If there is no desired restriction of an enviromental commodity tuple usage per timestep, the corresponding cell can be set to "inf" to ignore this parameter.

Maximum Annual Environmental Output, $\overline{M}_{vc}$, `m.commodity.loc[sit, com, com_type]['max']`: The parameter $\overline{M}_{vc}$ represents the maximum energy amount of an environmental commodity tuple c_{vq} ($\forall v \in V, q = "Env"$) 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. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{vq} and the fifth column of an environmental commodity tuples in this sheet with the header label "max" represents the parameter $\overline{M}_{vc}$. 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 Time Step, $\overline{g}_{vc}$, `m.commodity.loc[sit, com, com_type][`maxperstep`]`: The parameter $\overline{g}_{vc}$ represents the maximum energy amount of a sell commodity tuple c_{vq} ($\forall v \in V, q = "Sell"$) that energy model is allowed to sell per time step. 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 can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{vq} and the sixth column of sell commodity tuples in this sheet with the header label "maxperstep" represents the parameter $\overline{g}_{vc}$. 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, $\overline{G}_{vc}$, `m.commodity.loc[sit, com, com_type][`max`]`: The parameter $\overline{G}_{vc}$ represents the maximum energy amount of a sell commodity tuple c_{vq} ($\forall v \in V, q = "Sell"$) that energy model is allowed to sell annually. The unit of this parameter is MW. This parameter is to be provided by the user and to be entered in spreadsheet. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{vq} and the fifth column of sell commodity tuples in this sheet with the header label "max" represents the parameter $\overline{G}_{vc}$. 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 Time Step, $\overline{b}_{vc}$, `m.commodity.loc[sit, com, com_type][`maxperstep`]`: The parameter $\overline{b}_{vc}$ represents the maximum energy amount of a buy commodity tuple c_{vq} ($\forall v \in V, q = "Buy"$) that energy model is allowed to buy per time step. 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 can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{vq} and the sixth column of buy commodity tuples in this sheet with the header label "maxperstep" represents the parameter $\overline{b}_{vc}$. 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, $\overline{B}_{vc}$, `m.commodity.loc[sit, com, com_type][`max`]`: The parameter $\overline{B}_{vc}$ represents the maximum energy amount of a buy commodity tuple c_{vq} ($\forall v \in V, q = "Buy"$) that energy model is allowed to buy annually. The unit of this parameter is MW. This parameter is to be provided by the user and to be entered in spreadsheet. The related section for this parameter in the spreadsheet can be found under the `Commodity` sheet. Here each row represents another commodity tuple c_{vq} and the fifth column of buy commodity tuples in this sheet with the header label "max" represents the parameter $\overline{B}_{vc}$. 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$_2$ Emission Limit, $\overline{L}_{CO_2}$, `m.hack.loc['Global CO2 Limit', 'Value']`: The parameter $\overline{L}_{CO_2}$ represents the maximum total energy amount of all environmental commodities that energy model is allowed to produce and release to environment annually. This parameter is optional. If the user desires to set a maximum annual limit to total CO_2 emission of the whole energy model, this can be done by entering the desired value to the related spreadsheet. The related section for this parameter can be found under the sheet "hacks". Here the the cell where the "Global CO2 limit" row and "value" column intersects stands for the parameter $\overline{L}_{CO_2}$. 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, $\underline{K}_{vp}$, `m.process.loc[sit, pro]['cap-lo']`: The parameter $\underline{K}_{vp}$ represents the minimum amount of power output capacity of a process p at a site v, that energy model is allowed to have. 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 fourth column with the header label "cap-lo" represents the parameters $\underline{K}_{vp}$ 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", to ignore this parameter.

Process Capacity Installed, K_{vp}, `m.process.loc[sit, pro]['inst-cap']`: 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 simulation. 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 third 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, $\overline{K}_{vp}$, `m.process.loc[sit, pro]['cap-up']`: The parameter $\overline{K}_{vp}$ represents the maximum amount of power output capacity of a process p at a site v, that energy model is allowed to have. 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 fifth column with the header label "cap-up" represents the parameters $\overline{K}_{vp}$ of the corresponding process p and site v combinations. Alternatively, $\overline{K}_{vp}$ 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. If there is no desired maximum limit for the process capacities, both input parameters can be simply set to an unrealistic high value, to ignore this parameter.

Process Maximal Gradient, $\overline{PG}_{vp}$, `m.process.loc[sit, pro]['max-grad']`: The parameter $\overline{PG}_{vp}$ represents the maximal power gradient of a process p at a site v, 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 sixth column with the header label "max-grad" represents the parameters $\overline{PG}_{vp}$ of the corresponding process p and site v combinations. If there is no desired maximum limit for the process power gradient, this parameter can be simply set to an unrealistic high value, to ignore this parameter.

Process Minimum Part Load Fraction, $\underline{P}_{vp}$, `m.process.loc[sit, pro]['partial']`: The parameter $\underline{P}_{vp}$ represents the minimum allowable part load of a process p at a site v 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 twelfth column with the header label "partial" represents the parameters $\underline{P}_{vp}$ of the corresponding process p and site v combinations.

Process Input Ratio, r_{pc}^{in}, `m.r_in.loc[pro, co]`: The parameter r_{pc}^{in} represents the ratio of the input amount of a commodity c in a process p, relative to the process throughput at a given timestep. The related section for this parameter in the spreadsheet can be found under the "Process-Comodity" sheet. Here each row represents another commodity c that either goes in to or comes out of a process p. The fourth column with the header label "ratio" represents the parameters of the corresponding process p, commodity c and direction (In,Out) combinations.

Process Partial Input Ratio, $\underline{r}_{pc}^{\text{in}}$, `m.r_in_partial.loc[pro, co]`: The parameter $\underline{r}_{pc}^{\text{in}}$ represents the ratio of the amount of input commodity c a process p consumes if it is at its minimum allowable partial operation. More precisely, when its throughput τ_{vpt} has the minimum value $\omega_{vpt}\underline{P}_{vp}$.

Process Output Ratio, r_{pc}^{out}, `m.r_out.loc[pro, co]`: The parameter r_{pc}^{out} represents the ratio of the output amount of a commodity c in a process p, relative to the process throughput at a given timestep. The related section for this parameter in the spreadsheet can be found under the "Process-Comodity" sheet. Here each row represents another commodity c that either goes in to or comes out of a process p. The fourth column with the header label "ratio" represents the parameters of the corresponding process p, commodity c and direction (In,Out) combinations.

Process input and output ratios are, in general, dimensionless since the majority of output and input commodities are represented in MW. Exceptionally, some process input and output ratios can be assigned units e.g. the environmental commodity (`Env`) 'CO_2 could have a process output ratio with the unit of Mt/MWh.

Since process input and output ratios take the process throughput τ_{vpt} as the reference in order to calculate the input and output commodity flows, the process input (or output) ratio of "1" is assigned to the commodity which represents the throughput. By default, the major input commodity flow of the process (e.g. 'Gas' for 'Gas plant', 'Wind' for 'Wind park') represents the process throughput so those commodities have the process input (or output) ratio of "1"; but the "throughput" selection can be arbitrarily shifted to other commodities (e.g. power output of the process) by scaling all of the process input and output ratios by an appropriate factor.

Storage Technical Parameters

Initial and Final State of Charge (relative), I_{vs}, `m.storage.loc[sit, sto, com]['init']`: The parameter I_{vs} represents the initial load factor of a storage s in a site v. This parameter shows, as a percentage, how much of a storage is loaded at the beginning of the simulation. The same value should be preserved at the end of the simulation, to make sure that the optimization model doesn't consume the whole storage content at once and leave it empty at the end, otherwise this would disrupt the continuity of the optimization. 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 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 twentieth column with the header label "init" represents the parameters for corresponding storage s, site v, commodity c combinations.

Storage Efficiency During Charge, e_{vs}^{in}, `m.storage.loc[sit, sto, com]['eff-in']`: The parameter e_{vs}^{in} represents the charge efficiency of a storage s in a site v that stores a commodity c. The charge 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 charge with no power or energy loss and "0" represents that storage technology consumes whole enery during charge. 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 tenth column with the header label "eff-in" represents the parameters for corresponding storage s, site v, commodity c combinations.

Storage Efficiency During Discharge, e^out_{vs}, `m.storage.loc[sit, sto, com]['eff-out']`: The parameter e^out_{vs} represents the discharge efficiency of a storage s in a site v that stores a commodity c. The discharge efficiency shows, how much of a desired energy and accordingly power can be succesfully retrieved out of a storage. The value of this parameter is expressed as a normalized efipercentage, where "1" represents a discharge with no power or energy loss and "0" represents that storage technology consumes whole enery during discharge. 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 eleventh column with the header label "eff-out" represents the parameters for corresponding storage s, site v, commodity c combinations.

Storage Content Lower Bound, $\underline{K}^\text{c}_{vs}$, `m.storage.loc[sit, sto, com]['cap-lo-c']`: The parameter $\underline{K}^\text{c}_{vs}$ represents the minimum amount of energy content capacity allowed of a storage s storing a commodity c in a site v, that the energy system model is allowed to have. 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 fifth column with the header label "cap-lo-c" represents the parameters for corresponding storage s, site v, commodity c combinations. If there is no desired minimum limit for the storage energy content capacities, this parameter can be simply set to "0", to ignore this parameter.

Storage Content Installed, K^c_{vs}, `m.storage.loc[sit, sto, com]['inst-cap-c']`: The parameter K^c_{vs} represents the amount of energy content capacity of a storage s storing commodity c in a site v, that is already installed to the energy system at the beginning of the simulation. 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 fourth column with the header label "inst-cap-c" represents the parameters for corresponding storage s, site v, commodity c combinations.

Storage Content Upper Bound, $\overline{K}^\text{c}_{vs}$, `m.storage.loc[sit, sto, com]['cap-up-c']`: The parameter $\overline{K}^\text{c}_{vs}$ represents the maximum amount of energy content capacity allowed of a storage s storing a commodity c in a site v, that the energy system model is allowed to have. 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 sixth column with the header label "cap-up-c" represents the parameters for corresponding storage s, site v, commodity c combinations. If there is no desired maximum limit for the storage energy content capacitites, this parameter can be simply set to "”inf”" or an unrealistic high value, to ignore this parameter.

Storage Power Lower Bound, $\underline{K}^\text{p}_{vs}$, `m.storage.loc[sit, sto, com]['cap-lo-p']`: The parameter $\underline{K}^\text{p}_{vs}$ represents the minimum amount of power output capacity of a storage s storing commodity c in a site v, that energy system model is allowed to have. 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 eighth column with the header label "cap-lo-p" represents the parameters for corresponding storage s, site v, commodity c combinations. If there is no desired minimum limit for the storage energy content capacities, this parameter can be simply set to "0", to ignore this parameter.

Storage Power Installed, K^p_{vs}, `m.storage.loc[sit, sto, com]['inst-cap-p']`: The parameter K^c_{vs} represents the amount of power output capacity of a storage s storing commodity c in a site v, that is already installed to the energy system at the beginning of the simulation. 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 seventh column with the header label "inst-cap-p" represents the parameters for corresponding storage s, site v, commodity c combinations.

Storage Power Upper Bound, $\overline{K}_{vs}^{\text{p}}$, `m.storage.loc[sit, sto, com]['cap-up-p']`: The parameter $\overline{K}_{vs}^{\text{p}}$ represents the maximum amount of power output capacity allowed of a storage s storing a commodity c in a site v, that the energy system model is allowed to have. 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 sixth column with the header label "cap-up-p" represents the parameters for corresponding storage s, site v, commodity c combinations. If there is no desired maximum limit for the storage energy content capacitites, this parameter can be simply set to "”inf”" or an unrealistic high value, to ignore this parameter.

Transmission Technical Parameters

Transmission Efficiency, e_{af}, `m.transmission.loc[sin, sout, tra, com]['eff']`: The parameter e_{af} represents the energy efficiency of a transmission f that transfers a commodity c through an arc a. 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 can be found under the "Transmission" sheet. Here each row represents another transmission,site in, site out, commodity combination. The fifth column with the header label "eff" represents the parameters e_{af} of the corresponding combinations.

Transmission Capacity Lower Bound, $\underline{K}_{af}$, `m.transmission.loc[sin, sout, tra, com]['cap-lo']`: 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 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 can be found under the "Transmission" sheet. Here each row represents another transmission,site in, site out, commodity combination. The tenth column with the header label "cap-lo" represents the parameters $\underline{K}_{af}$ of the corresponding combinations.

Transmission Capacity Installed, K_{af}, `m.transmission.loc[sin, sout, tra, com]['inst-cap']`: 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 simulation. 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,site in, site out, commodity combination. The tenth column with the header label "inst-cap" represents the parameters K_{af} of the corresponding combinations.

Transmission Capacity Upper Bound, $\overline{K}_{af}$, `m.transmission.loc[sin, sout, tra, com]['cap-up']`: The parameter $\overline{K}_{af}$ represents the maximum power output capacity of a transmission f transferring a commodity c through an arc a, 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 can be found under the "Transmission" sheet. Here each row represents another transmission, site in, site out, commodity combination. The tenth column with the header label "cap-up" represents the parameters $\overline{K}_{af}$ of the corresponding combinations.

Demand Side Management Technical Parameters

DSM Efficiency, e_{vc}, `m.dsm.loc[sit, com]['eff']`: The parameter e_{vc} represents the efficiency of the DSM upshift process. Which means losses of the DSM up- or downshift have to be taken into account by this factor.

DSM Delay Time, y_{vc}, `m.dsm.loc[sit, com]['delay']`: The delay time y_{vc} restricts how long the time delta between an upshift and its corresponding downshifts may be.

DSM Recovery Time, o_{vc}, `m.dsm.loc[sit, com]['recov']`: The recovery time o_{vc} prevents the DSM system to continously shift demand. During the recovery time, all upshifts may not exceed a predfined value.

DSM Maximal Upshift Capacity, $\overline{K}_{vc}^{\text{up}}$, MW, `m.dsm.loc[sit, com]['cap-max-up']`: The DSM upshift capacity $\overline{K}_{vc}^{\text{up}}$ limits the total upshift in one time step.

DSM Maximal Downshift Capacity, $\overline{K}_{vc}^{\text{down}}$, MW, `m.dsm.loc[sit, com]['cap-max-down']`: Correspondingly, the DSM downshift capacity $\overline{K}_{vc}^{\text{down}}$ limits the total downshift in one time step.

Economical Parameters

Table 13: *Table: Economical Model Parameters*

Parameter	Unit	Description
AF	_	Annuity factor
Commodity Economical Parameters		
k_{vc}^{fuel}	€/MWh	Stock Commodity Fuel Costs
k_{vc}^{env}	€/MWh	Environmental Commodity Costs
k_{vct}^{bs}	€/MWh	Buy/Sell Commodity Buy/Sell Costs
Process Economical Parameters		
i_{vp}	_	Weighted Average Cost of Capital for Process
z_{vp}	_	Process Depreciation Period
k_{vp}^{inv}	€/(MW a)	Annualised Process Capacity Investment Costs
k_{vp}^{fix}	€/(MW a)	Process Capacity Fixed Costs
k_{vp}^{var}	€/MWh	Process Variable Costs
k_{vp}^{st}	€	Process Startup Costs
Storage Economical Parameters		
i_{vs}	_	Weighted Average Cost of Capital for Storage
z_{vs}	_	Storage Depreciation Period
$k_{vs}^{\text{p,inv}}$	€/(MWh a)	Annualised Storage Power Investment Costs
$k_{vs}^{\text{p,fix}}$	€/(MW a)	Annual Storage Power Fixed Costs
$k_{vs}^{\text{p,var}}$	€/MWh	Storage Power Variable Costs
$k_{vs}^{\text{c,inv}}$	€/(MWh a)	Annualised Storage Size Investment Costs
$k_{vs}^{\text{c,fix}}$	€/(MWh a)	Annual Storage Size Fixed Costs
$k_{vs}^{\text{c,var}}$	€/MWh	Storage Usage Variable Costs
Transmission Economical Parameters		
i_{vf}	_	Weighted Average Cost of Capital for Transmission
z_{af}	_	Tranmission Depreciation Period
k_{af}^{inv}	€/(MW a)	Annualised Transmission Capacity Investment Costs
k_{af}^{fix}	€/(MWh a)	Annual Transmission Capacity Fixed Costs
k_{af}^{var}	€/MWh	Tranmission Usage Variable Costs

Annuity factor, $AF(n, i)$,: Annuity factor AF is used to calculate the present value of future fixed annuities. The parameter annuity factor is the only parameter that is not given as an input by the user. This parameter is derived from the parameters WACC i (Weighted average cost of capital) and Depreciation z by the annuity factor formula. The value of this parameter is expressed with the following equation.

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

where;

- n represents the depreciation period z.
- i represents the weighted average cost of capital(wacc) i.

This derived parameter is calculated by the helper function `annuity factor()` and defined by the following code fragment.

```
# derive annuity factor from WACC and depreciation periods
process['annuity-factor'] = annuity_factor(
```

(continues on next page)

(continued from previous page)

```
    process['depreciation'], process['wacc'])
transmission['annuity-factor'] = annuity_factor(
    transmission['depreciation'], transmission['wacc'])
storage['annuity-factor'] = annuity_factor(
    storage['depreciation'], storage['wacc'])
```

`urbs.`**`annuity_factor`**`()`

Annuity factor formula.

Evaluates the annuity factor formula for depreciation duration and interest rate. Works also well for equally sized numpy arrays as input.

Parameters

- **n** (*int*) – number of depreciation periods (years)
- **i** (*float*) – interest rate (e.g. 0.06 means 6 %)

Returns value of the expression $\frac{(1+i)^n i}{(1+i)^n - 1}$

Commodity Economical Parameters

Stock Commodity Fuel Costs, k_{vc}^{fuel}, `m.commodity.loc[c]['price']`: The parameter k_{vc}^{fuel} represents the purchase cost for purchasing one unit (1 MWh) of a stock commodity c ($\forall c \in C_{\text{stock}}$) in a site v ($\forall v \in V$). The unit of this parameter is €/MWh. The related section for this parameter in the spreadsheet can be found under the "Commodity" sheet. Here each row represents another commodity tuple c_{vq} and the fourth column of stock commodity tuples ($\forall q = "Stock"$) in this sheet with the header label "price" represents the corresponding parameter k_{vc}^{fuel}.

Environmental Commodity Costs, k_{vc}^{env}, `m.commodity.loc[c]['price']`: The parameter k_{vc}^{env} represents the cost for producing/emitting one unit (1 t, 1 kg, . . .) of an environmentcal commodity c ($\forall c \in C_{\text{env}}$) 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 is the "Commodity" sheet. Here, each row represents a commodity tuple c_{vq} and the fourth column of environmental commodity tuples ($\forall q = "Env"$) in this sheet with the header label "price" represents the corresponding parameter k_{vc}^{env}.

Buy/Sell Commodity Buy/Sell Costs, k_{vct}^{bs}, `com_prices[c].loc[tm]`: The parameter k_{vct}^{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 a site v ($\forall v \in V$) at a 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 under the "Commodity" sheet. Here each row represents another commodity tuple c_{vq} and the fourth column of buy/sell commodity tuples ($\forall q = "Buy"$)/($\forall q = "Sell"$) in this sheet with the header label "price" represents how the parameter k_{vct}^{bs} will be defined. There are two options for this parameter. This parameter will either be a fix value for the whole simulation duration or will vary with the timesteps t. For the first option, if the buy/sell price of a buy/sell commodity is a fix value for the whole simulation duration, this value can be entered directly into the corresponding cell with the unit €/MWh. For the second option, if the buy/sell price of a buy/sell commodity depends on time, accordingly on timesteps, a string (a linear sequence of characters, words, or other data) should be written in the corresponding cell. An example string looks like this: "1,25xBuy" where the first numbers (1,25) represent a coefficient for the price. This value is than multiplied by values from another list given with timeseries. Here the word "Buy" refers to a timeseries located in ""Buy-Sell-Price"" sheet with commodity names, types and timesteps. This timeseries should be filled with time dependent buy/sell price variables. The parameter k_{vct}^{bs} is then calculated by the product of the price coefficient and the related time variable for a given timestep t. This calculation and the decision for one of the two options is executed by the helper function `get_com_price()`.

`urbs.`**`get_com_price`**`(`*instance*, *tuples*`)`

Parameters

- **`instance`** (`str`) – a Pyomo ConcreteModel instance
- **`tuples`** (`list`) – a list of (site, commodity, commodity type) tuples

Returns a Pandas DataFrame with entities as columns and timesteps as index

Calculate commodity prices for each modelled timestep. Checks whether the input is a float. If it is a float it gets the input value as a fix value for commodity price. Otherwise if the input value is not a float, but a string, it extracts the price coefficient from the string and multiplies it with a timeseries of commodity price variables.

Process Economical Parameters

Weighted Average Cost of Capital for Process, i_{vp}, : The parameter i_{vp} represents the weighted average cost of capital for a process technology p 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 can be found under the "Process" sheet. Here each row represents another process p in a site v and the tenth column with the header label "wacc" represents the parameters i_{vp} of the corresponding process p and site v combinations. The parameter is given as a percentage, where "0,07" means 7%

Process Depreciation Period, z_{vp}, (a): The parameter z_{vp} represents the depreciation period of a process p in a site v. The depreciation period gives the economic lifetime (more conservative than technical lifetime) of a process investment. 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 p in a site v and the eleventh column with the header label "depreciation" represents the parameters z_{vp} of the corresponding process p and site v combinations.

Annualised Process Capacity Investment Costs, k_{vp}^{inv}, `m.process.loc[p]['inv-cost'] * m.process.loc[p]['annuity-factor']`: The parameter k_{vp}^{inv} represents the annualised investment cost for adding one unit new capacity of a process technology p in a site v. The unit of this parameter is €/(MW a). This parameter is derived by the product of annuity factor AF and the process capacity investment cost for a given process tuple. 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 can be found under the "Process" sheet. Here each row represents another process p in a site v and the seventh 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_{vp}^{fix}, `m.process.loc[p]['fix-cost']`: The parameter k_{vp}^{fix} represents the fix cost per one unit capacity κ_{vp} of a process technology p 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 can be found under the "Process" sheet. Here each row represents another process p in a site v and the eighth column with the header label "fix-cost" represents the parameters k_{vp}^{fix} of the corresponding process p and site v combinations.

Process Variable Costs, k_{vp}^{var}, `m.process.loc[p]['var-cost']`: The parameter k_{vp}^{var} represents the variable cost per one unit energy throughput τ_{vpt} through a process technology p in a site v. The unit of this parameter is €/MWh. 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 ninth column with the header label "var-cost" represents the parameters k_{vp}^var of the corresponding process p and site v combinations.

Process Startup Costs, k_{vp}^st, `m.process.loc[p]['startup']`: The parameter k_{vp}^st represents the startup cost per "startup occurence" of a process technology p in a site v. The unit of this parameter is €. 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 thirteenth column with the header label "startup" represents the parameters k_{vp}^st of the corresponding process p and site v combinations.

Storage Economical Parameters

Weighted Average Cost of Capital for Storage, i_{vs}, : The parameter i_{vs} represents the weighted average cost of capital for a storage technology s 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 can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the nineteenth column with the header label "wacc" represents the parameters i_{vs} 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_{vs}, (a): The parameter z_{vs} represents the depreciation period of a storage s in a site v. The depreciation period gives the economic lifetime (more conservative than technical lifetime) of a storage investment. 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 eighteenth column with the header label "depreciation" represents the parameters z_{vs} of the corresponding storage s and site v combinations.

Annualised Storage Power Investment Costs, $k_{vs}^\text{p,inv}$, `m.storage.loc[s]['inv-cost-p'] * m.storage.loc[s]['annuity-factor']`: The parameter $k_{vs}^\text{p,inv}$ represents the annualised investment cost for adding one unit new power output capacity of a storage technology s in a site v. The unit of this parameter is €/(MWh a). This parameter is derived by the product of annuity factor AF and the investment cost for one unit of new power output capacity of a storage s in a site v, which is to be given as an input parameter by the user. The related section for the storage power output capacity investment cost in the spreadsheet can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the twelfth 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, $k_{vs}^\text{p,fix}$, `m.storage.loc[s]['fix-cost-p']`: The parameter $k_{vs}^\text{p,fix}$ represents the fix cost per one unit power output capacity of a storage technology s 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 can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the fourteenth column with the header label "fix-cost-p" represents the parameters $k_{vs}^\text{p,fix}$ of the corresponding storage s and site v combinations.

Storage Power Variable Costs, $k_{vs}^\text{p,var}$, `m.storage.loc[s]['var-cost-p']`: The parameter $k_{vs}^\text{p,var}$ represents the variable cost per unit energy, that is stored in or retrieved from a storage technology s in a site v. 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 sixteenth column with the header label "var-cost-p" represents the parameters $k_{vs}^\text{p,var}$ of the corresponding storage s and site v combinations.

Annualised Storage Size Investment Costs, $k_{vs}^\text{c,inv}$, `m.storage.loc[s]['inv-cost-c'] * m.storage.loc[s]['annuity-factor']`: The parameter $k_{vs}^\text{c,inv}$ represents the annualised investment cost for adding one unit new storage capacity to a storage technology s in a site v. The unit of this parameter is €/(MWh a). This parameter is derived by the product of annuity factor AF and the investment cost for one unit of new storage capacity of a storage s in a site v, which is to be given as an input parameter by the user. The related section for the storage content capacity investment cost in the spreadsheet can be found under the "Storage" sheet. Here each row represents another storage s in a site v and the thirteenth 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, $k_{vs}^{\text{c,fix}}$, `m.storage.loc[s]['fix-cost-c']`: The parameter $k_{vs}^{\text{c,fix}}$ represents the fix cost per one unit storage content capacity of a storage technology s in a site v, that is charged annually. The unit of this parameter is €/(MWh a). 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 fifteenth column with the header label "fix-cost-c" represents the parameters $k_{vs}^{\text{c,fix}}$ of the corresponding storage s and site v combinations.

Storage Usage Variable Costs, $k_{vs}^{\text{c,var}}$, `m.storage.loc[s]['var-cost-c']`: The parameter $k_{vs}^{\text{p,var}}$ represents the variable cost per unit energy, that is conserved in a storage technology s in a site v. 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 seventeenth column with the header label "var-cost-c" represents the parameters $k_{vs}^{\text{c,var}}$ 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 Economical Parameters

Weighted Average Cost of Capital for Transmission, i_{vf}, : The parameter i_{vf} represents the weighted average cost of capital for a transmission f transferring commodities through an arc a. 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 can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the twelfth column with the header label "wacc" represents the parameters i_{vf} 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_{af}, (a): The parameter z_{af} represents the depreciation period of a transmission f transferring commodities through an arc a. The depreciation period of gives the economic lifetime (more conservative than technical lifetime) of a transmission investment. The unit of this parameter is €/ (MW a). The related section for this parameter in the spreadsheet can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the thirteenth column with the header label "depreciation" represents the parameters z_{af} of the corresponding transmission f and arc a combinations.

Annualised Transmission Capacity Investment Costs, k_{af}^{inv}, `m.transmission.loc[t]['inv-cost'] * m.transmission.loc[t]['annuity-factor']`: The parameter k_{af}^{inv} represents the annualised investment cost for adding one unit new transmission capacity to a transmission f transferring commodities through an arc a. This parameter is derived by the product of annuity factor AF and the investment cost for one unit of new transmission capacity of a transmission f running through an arc a, which is to be given as an input parameter by the user. The unit of this parameter is €/(MW a). The related section for the transmission capacity investment cost in the spreadsheet can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the sixth 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_{af}^{fix}, `m.transmission.loc[t]['fix-cost']`: The parameter k_{af}^{fix} represents the fix cost per one unit capacity of a transmission f transferring commodities through an arc a, that is charged annually. The unit of this parameter is €/(MWh a). The related section for this parameter in the spreadsheet can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the seventh column with the header label "fix-cost" represents the parameters k_{af}^{fix} of the corresponding transmission f and arc a combinations.

Transmission Usage Variable Costs, k_{af}^{var}, `m.transmission.loc[t]['var-cost']`: The parameter k_{af}^{var} represents the variable cost per unit energy, that is transferred with a transmissiom f through an arc a. The unit of this parameter is €/ MWh. The related section for this parameter in the spreadsheet can be found under the "Transmission" sheet. Here each row represents another transmission f transferring commodities through an arc a and the eighth column with the header label "var-cost" represents the parameters k_{af}^{var} of the corresponding transmission f and arc a combinations.

Equations

Objective function

The variable total system cost ζ is 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:

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

The calculation of the variable total system cost is given in `urbs.py` by the following code fragment.

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`. The calculation of each single cost types are listed below.

Investment Costs

The variable investment costs ζ_{inv} represent the required annual expenses made, in the hope of future benefits. These expenses are made on every new investment. The possible investments of an energy system in this model are:

1. Additional throughput capacity for process technologies.
2. Additional power capacity for storage technologies and additional storage content capacity for storage technologies.
3. Additional power capacity for transmission technologies.

The calculation of total annual investment cost ζ_{inv} is expressed by the formula:

$$\zeta_{\text{inv}} = \sum_{\substack{v\in V\\ p\in P}} \hat{\kappa}_{vp} k_p^{\text{inv}} + \sum_{\substack{v\in V\\ s\in S}} \left(\hat{\kappa}_{vs}^{\text{c}} k_{vs}^{\text{c,inv}} + \hat{\kappa}_{vs}^{\text{p}} k_{vs}^{\text{p,inv}} \right) + \sum_{\substack{a\in A\\ f\in F}} \hat{\kappa}_{af} k_{af}^{\text{inv}}$$

Total annual investment cost is calculated by the sum of three main summands, these are the investment costs for processes, storages, and transmissions.

1. The first summand of the formula calculates the required annual investment expenses to install the additional process capacity for every member of the set process tuples $\forall p_v \in P_v$. Total process investment cost for all process tuples is defined by the sum of all possible annual process investment costs, which are calculated seperately for each process tuple (p_v, `m.pro_tuples`) consisting of process p in site v. Annual process investment cost for a given process tuple p_v is calculated by the product of the variable new process capacity ($\hat{\kappa}_{vp}$,``m.cap_pro_new``) and the parameter annualised process capacity investment cost (k_{vp}^{inv}, `m.process.loc[p]['inv-cost'] * m.process.loc[p]['annuity-factor']`). In mathematical notation this summand is expressed as:

$$\sum_{\substack{v \in V\\ p \in P}} \hat{\kappa}_{vp} k_p^{\text{inv}}$$

2. The second summand of the formula calculates the required investment expenses to install additional power output capacity and storage content capacity to storage technologies for every member of the set storage tuples ($\forall s_{vc} \in S_{vc}$). This summand consists of two products:

   - The first product calculates the required annual investment expenses to install an additional storage content capacity to a given storage tuple . This is calculated by the product of the variable new storage size ($\hat{\kappa}_{vs}^{\text{c}}$, `cap_sto_c_new`) and the parameter annualised storage size investment costs ($k_{vs}^{\text{c,inv}}$, `m.storage.loc[s]['inv-cost-c'] * m.storage.loc[s]['annuity-factor']`).
   - The second product calculates the required annual investment expenses to install an additional power output capacity to a given storage tuple. This is calculated by the product of the variable new storage power ($\hat{\kappa}_{vs}^{\text{p}}$, `cap_sto_p_new`) and the parameter annualised storage power investment costs ($k_{vs}^{\text{p,inv}}$, `m.storage.loc[s]['inv-cost-p'] * m.storage.loc[s]['annuity-factor']`).

   These two products for a given storage tuple are than added up. The calculation of investment costs for a given storage tuple is than repeated for every single storage tuple and summed up to calculate the total investment costs for storage technologies. In mathematical notation this summand is expressed as:

$$\sum_{\substack{v \in V\\ s \in S}} (\hat{\kappa}_{vs}^{\text{c}} k_{vs}^{\text{c,inv}} + \hat{\kappa}_{vs}^{\text{p}} k_{vs}^{\text{p,inv}})$$

3. The third and the last summand of the formula calculates the required investment expenses to install additional power capacity to transmission technologies for every member of the set transmission tuples $\forall f_{ca} \in F_{ca}$. Total transmission investment cost for all transmission tuples is defined by the sum of all possible annual transmission investment costs, which are calculated seperately for each transmission tuple (f_{ca}). Annual transmission investment cost for a given transmission tuple is calculated by the product of the variable new transmission capacity ($\hat{\kappa}_{af}$, `cap_tra_new`) and the parameter annualised transmission capacity investment costs (k_{af}^{inv}, `m.transmission.loc[t]['inv-cost'] * m.transmission.loc[t]['annuity-factor']`) for the given transmission tuple. In mathematical notation this summand is expressed as:

$$\sum_{\substack{a \in A\\ f \in F}} \hat{\kappa}_{af} k_{af}^{\text{inv}}$$

As mentioned above the variable investment costs ζ_{inv} is calculated by the sum of these 3 summands.

In script `urbs.py` the value of the total investment cost is calculated by the following code fragment:

```
if cost_type == 'Invest':
    return m.costs[cost_type] == \
        sum(m.cap_pro_new[p] *
            m.process.loc[p]['inv-cost'] *
            m.process.loc[p]['annuity-factor']
            for p in m.pro_tuples) + \
        sum(m.cap_tra_new[t] *
            m.transmission.loc[t]['inv-cost'] *
            m.transmission.loc[t]['annuity-factor']
            for t in m.tra_tuples) + \
        sum(m.cap_sto_p_new[s] *
            m.storage.loc[s]['inv-cost-p'] *
            m.storage.loc[s]['annuity-factor'] +
            m.cap_sto_c_new[s] *
            m.storage.loc[s]['inv-cost-c'] *
            m.storage.loc[s]['annuity-factor']
            for s in m.sto_tuples)
```

Fix Costs

The variable fix costs ζ_{fix} represents the total annual fixed costs for all used storage, process and transmission technologies. The possible fix costs of an energy system in this model can be divided into sections, these are:

1. Fix costs for process technologies
2. Fix costs for storage technologies
3. Fix costs for transmission technologies.

The calculation of total annual fix cost ζ_{fix} is expressed by the formula:

$$\zeta_{\text{fix}} = \sum_{\substack{v\in V\\ p\in P}} \kappa_{vp} k_{vp}^{\text{fix}} + \sum_{\substack{v\in V\\ s\in S}} \left(\kappa_{vs}^{\text{c}} k_{vs}^{\text{c,fix}} + \kappa_{vs}^{\text{p}} k_{vs}^{\text{p,fix}}\right) + \sum_{\substack{a\in A\\ f\in F}} \kappa_{af} k_{af}^{\text{fix}}$$

Total annual fix cost ζ_{fix} is calculated by the sum of three main summands, these are the fix costs for process, storage and transmission technologies.

1. The first summand of the formula calculates the required annual fix cost to keep all the process technologies maintained. This is calculated for every member of the set process tuples $\forall p_v \in P_v$. Total process fix cost for all process tuples is defined by the sum of all possible annual process fix costs, which are calculated seperately for each process tuple (p_v, `m.pro_tuples`) consisting of process p in site v. Annual process fix cost for a given process tuple is calculated by the product of the variable total process capacity (κ_{vp}, `cap_pro`) and process capacity fixed cost (k_{vp}^{fix}, `m.process.loc[p]['fix-cost']`). In mathematical notation this summand is expressed as:

$$\sum_{\substack{v\in V\\ p\in P}} \kappa_{vp} k_{vp}^{\text{fix}}$$

2. The second summand of the formula calculates the required fix expenses to keep the power capacity and storage content capacity of storage technologies maintained. The present storage technologies comprise the members of the set storage tuples $\forall s_{vc} \in S_{vc}$. This summand consists of two products:

- The first product calculates the required annual fix expenses to keep the storage content capacity of a given storage tuple maintained. This is calculated by the product of the variable total storage size (κ_{vs}^{c}, `cap_sto_c`) and the parameter annual storage size fixed costs ($k_{vs}^{\text{c,fix}}$, `m.storage.loc[s]['fix-cost-c']`).
- The second product calculates the required annual fix expenses to keep the power capacity of a given storage tuple maintained. This is calculated by the product of the variable total storage power (κ_{vs}^{p}, `cap_sto_p`) and the parameter annual storage power fixed costs ($k_{vs}^{\text{p,fix}}$, `m.storage.loc[s]['fix-cost-p']`).

These two products for a given storage tuple are than added up. The calculation of fix costs for a storage tuple is then repeated for every single storage tuple and summed up to calculate the total fix costs for storage technologies. In mathematical notation this summand is expressed as:

$$\sum_{\substack{v\in V\\ s\in S}} (\kappa_{vs}^{\text{c}} k_{vs}^{\text{c,fix}} + \kappa_{vs}^{\text{p}} k_{vs}^{\text{p,fix}})$$

3. The third and the last summand of the formula calculates the required fix expenses to keep the power capacity of transmission technologies maintained. The transmission technologies comprise the members of the set transmission tuples $\forall f_{ca} \in F_{ca}$. Total transmission fix cost for all transmission tuples is defined by the sum of all possible annual transmission fix costs, which are calculated seperately for each transmission tuple f_{ca}. Annual transmission fix cost for a given transmission tuple is calculated by the product of the variable total transmission capacity (κ_{af}, `cap_tra`) and the parameter annual transmission capacity fixed costs (k_{af}^{fix}, `m.transmission.loc[t]['fix-cost']`) for the given transmission tuple f_{ca}. In mathematical notation this summand is expressed as:

$$\sum_{\substack{a\in A\\ f\in F}} \kappa_{af} k_{af}^{\text{fix}}$$

As mentioned above, the fix costs ζ_{fix} are calculated by the sum of these 3 summands.

In script `urbs.py` the value of the total fix cost is calculated by the following code fragment:

```python
elif cost_type == 'Fixed':
    return m.costs[cost_type] == \
        sum(m.cap_pro[p] * m.process.loc[p]['fix-cost']
            for p in m.pro_tuples) + \
        sum(m.cap_tra[t] * m.transmission.loc[t]['fix-cost']
            for t in m.tra_tuples) + \
        sum(m.cap_sto_p[s] * m.storage.loc[s]['fix-cost-p'] +
            m.cap_sto_c[s] * m.storage.loc[s]['fix-cost-c']
            for s in m.sto_tuples)
```

Variable Costs

$$\zeta_{\text{var}} = w \sum_{t \in T_{\text{m}}} \left(\sum_{\substack{v \in V \\ p \in P}} \tau_{vpt} k_{vp}^{\text{var}} \Delta t + \sum_{\substack{a \in a \\ f \in F}} \pi_{af}^{\text{in}} k_{af}^{\text{var}} \Delta t + \right.$$

$$\left. \sum_{\substack{v \in V \\ s \in S}} \left[\epsilon_{vst}^{\text{con}} k_{vs}^{\text{c,var}} + \left(\epsilon_{vst}^{\text{in}} + \epsilon_{vst}^{\text{out}} \right) k_{vs}^{\text{p,var}} \Delta t \right] \right)$$

```
elif cost_type == 'Variable':
    return m.costs[cost_type] == \
        sum(m.tau_pro[(tm,) + p] * m.dt *
            m.process.loc[p]['var-cost'] *
            m.weight
            for tm in m.tm
            for p in m.pro_tuples) + \
        sum(m.e_tra_in[(tm,) + t] * m.dt *
            m.transmission.loc[t]['var-cost'] *
            m.weight
            for tm in m.tm
            for t in m.tra_tuples) + \
        sum(m.e_sto_con[(tm,) + s] *
            m.storage.loc[s]['var-cost-c'] * m.weight +
            (m.e_sto_in[(tm,) + s] + m.e_sto_out[(tm,) + s]) * m.dt *
            m.storage.loc[s]['var-cost-p'] * m.weight
            for tm in m.tm
            for s in m.sto_tuples)
```

Fuel Costs

The variable fuel costs ζ_{fuel} represents the total annual expenses that are required to be made to buy stock commodities $c \in C_{\text{stock}}$. The calculation of the variable total annual fuel cost ζ_{fuel} is expressed by the following mathematical notation:

$$\zeta_{\text{fuel}} = w \sum_{t \in T_{\text{m}}} \sum_{v \in V} \sum_{c \in C_{\text{stock}}} \rho_{vct} k_{vc}^{\text{fuel}} \Delta t$$

The variable ζ_{fuel} is calculated by the sum of all possible annual fuel costs, defined by the combinations of commodity tuples of commodity type 'Stock'($\forall c_{vq} \in C_{vq} \land q =$ 'Stock') and timesteps($\forall t \in T_m$). These annual fuel costs are calculated by the product of the following elements:

- The parameter stock commodity fuel cost for a given stock commodity c in a site v.(k_{vc}^{fuel}, `m.commodity.loc[c]['price']`)
- The variable stock commodity source term for a given stock commodity c in a site v at a timestep t.(ρ_{vct}, `e_co_stock`)
- The variable timestep duration.(Δt, `dt`)
- The variable weight.(w, `weight`)

In script `urbs.py` the value of the total fuel cost is calculated by the following code fragment:

```
elif cost_type == 'Fuel':
    return m.costs[cost_type] == sum(
        m.e_co_stock[(tm,) + c] * m.dt *
        m.commodity.loc[c]['price'] *
        m.weight
        for tm in m.tm for c in m.com_tuples
        if c[1] in m.com_stock)
```

Revenue Costs

The variable revenue costs ζ_rev represents the total annual expenses that are required to be made to sell sell commodities $c \in C_\text{sell}$. The calculation of the variable total annual revenue cost ζ_rev is expressed by the following mathematical notation:

$$\zeta_\text{rev} = -w \sum_{t\in T_\text{m}} \sum_{v\in V} \sum_{c\in C_\text{sell}} \varrho_{vct} k_{vct}^\text{bs} \Delta t$$

The variable ζ_rev is calculated by the sum of all possible annual revenue costs, defined by the combinations of commodity tuples of commodity type 'Sell'($\forall c_{vq} \in C_{vq} \land q =$ 'Sell') and timesteps ($\forall t \in T_m$). These annual revenue costs are calculated by the product of the following elements:

- The parameter sell commodity sell cost for given sell commodity c in a site v.(k_{vct}^bs, `com_prices[c].loc[tm]`)
- The variable sell commodity source term for a given sell commodity c in a site v at a timestep t.(ϱ_{vct}, `e_co_sell`)
- The variable timestep duration.(Δt, `dt`)
- The variable weight.(w, `weight`)
- Coefficient [-1].

Since this variable is an income for the energy system, it is multiplied by the value -1 to be able to express it in the cost function as a summand. In script `urbs.py` the value of the total revenue cost is calculated by the following code fragment:

```
elif cost_type == 'Revenue':
    sell_tuples = commodity_subset(m.com_tuples, m.com_sell)
    com_prices = get_com_price(m, sell_tuples)

    return m.costs[cost_type] == -sum(
        m.e_co_sell[(tm,) + c] *
        com_prices[c].loc[tm] *
        m.weight * m.dt
        for tm in m.tm
        for c in sell_tuples)
```

Purchase Costs

The variable purchase costs ζ_pur represents the total annual expenses that are required to be made to purchase buy commodities $c \in C_\text{buy}$. The calculation of the variable total annual purchase cost ζ_pur is expressed by the following mathematical notation:

$$\zeta_{\text{pur}} = w \sum_{t \in T_{\text{m}}} \sum_{v \in V} \sum_{c \in C_{\text{buy}}} \psi_{vct} k_{vct}^{\text{bs}} \Delta t$$

The variable ζ_{pur} is calculated by the sum of all possible annual purchase costs, defined by the combinations of commodity tuples of commodity type 'Buy'($\forall c_{vq} \in C_{vq} \land q =$ 'Buy') and timesteps ($\forall t \in T_m$). These annual purchase costs are calculated by the product of the following elements:

- The parameter buy commodity buy cost for a given buy commodity c in a site v. (k_{vct}^{bs}, `com_prices[c].loc[tm]`)
- The variable buy commodity source term for a given buy commodity c in a site v at a timestep t.(ψ_{vct}, `e_co_buy`)
- The variable timestep duration.(Δt, `dt`)
- The variable weight.(w, `weight`)

In script `urbs.py` the value of the total purchase cost is calculated by the following code fragment:

```
elif cost_type == 'Purchase':
    buy_tuples = commodity_subset(m.com_tuples, m.com_buy)
    com_prices = get_com_price(m, buy_tuples)

    return m.costs[cost_type] == sum(
        m.e_co_buy[(tm,) + c] *
        com_prices[c].loc[tm] *
        m.weight * m.dt
        for tm in m.tm
        for c in buy_tuples)
```

Startup Costs

The variable startup costs ζ_{startup} represents the total annual expenses that are required for the startup occurences of processes with the partial & startup feature activated. The calculation of the total annual startup costs is expressed by the following mathematical notation:

$$\zeta_{\text{startup}} = w \sum_{t \in T_{\text{m}}} \sum_{v \in V} \sum_{p \in P} \phi_{vpt} k_{vp}^{\text{st}} \Delta t$$

In script `urbs.py` the value of the total startup cost is calculated by the following code fragment:

```
elif cost_type == 'Startup':
    return m.costs[cost_type] == sum(
        m.startup_pro[(tm,) + p] *
        m.process.loc[p]['startup-cost'] *
        m.weight * m.dt
        for tm in m.tm
        for p in m.pro_partial_tuples)
```

Environmental Costs

Environmental costs ζ_{env} represent the total annual taxes for created emissions/pollutions in form of environmental commodities. The total annual costs are calculated by summing the negative commodity balance CB of all environmental commodities, multiplied by their respective price

$$\zeta_{\text{env}} = -w \sum_{t\in T_{\text{m}}} \sum_{v\in V} \sum_{c\in C_{\text{env}}} \text{CB}(v,c,t)\Delta t$$

In script `urbs.py` the value of the total environmental cost is calculated by the following code fragment:

```
elif cost_type == 'Environmental':
    return m.costs[cost_type] == sum(
        - commodity_balance(m, tm, sit, com) *
        m.weight * m.dt *
        m.commodity.loc[sit, com, com_type]['price']
        for tm in m.tm
        for sit, com, com_type in m.com_tuples
        if com in m.com_env)
```

Constraints

Commodity Constraints

Commodity Balance The function commodity balance calculates the balance of a commodity c in a site v at a timestep t. Commodity balance function facilitates the formulation of commodity constraints. The formula for commodity balance is expressed in mathematical notation as:

$$\text{CB}(v,c,t) = \sum_{p|c\in C_{vp}^{\text{in}}} \epsilon_{vcpt}^{\text{in}} - \sum_{p|c\in C_{vp}^{\text{out}}} \epsilon_{vcpt}^{\text{out}} + \sum_{s\in S_{vc}} \left(\epsilon_{vst}^{\text{in}} - \epsilon_{vst}^{\text{out}}\right) + \sum_{\substack{a\in A_v^{\text{s}} \\ f\in F_{vc}^{\text{exp}}}} \pi_{aft}^{\text{in}} - \sum_{\substack{a\in A_v^{\text{p}} \\ f\in F_{vc}^{\text{imp}}}} \pi_{aft}^{\text{out}}$$

This function sums up for a given commodity c, site v and timestep t;

- the consumption: Process input commodity flow $\epsilon_{vcpt}^{\text{in}}$ of all process tuples using the commodity c in the site v at the timestep t.
- the export: Input transmission power flow π_{aft}^{in} of all transmission tuples exporting the commodity c from the origin site v at the timestep t.
- the storage input: Input power flow $\epsilon_{vst}^{\text{in}}$ of all storage tuples storing the commodity c in the site v at the timestep t.

and subtracts for the same given commodity c, site v and timestep t;

- the creation: Process output commodity flow $\epsilon_{vcpt}^{\text{out}}$ of all process tuples using the commodity c in the site v at the timestep t.
- the import: Output transmission power flow π_{aft}^{out} of all transmission tuples importing the commodity math:c to the destination site v at the timestep t.
- the storage output: Output power flow $\epsilon_{vst}^{\text{out}}$ of all storage tuples storing the commodity c in the site v at the timestep t.

The value of the function CB being greater than zero $\text{CB} > 0$ means that the presence of the commodity c in the site v at the timestep t is getting less than before by 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.

In script `urbs.py` the value of the commodity balance function $\mathrm{CB}(v, c, t)$ is calculated by the following code fragment:

Vertex Rule: Vertex rule is the main constraint that has to be satisfied for every commodity. 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 at a timestep t to be balanced by a corresponding source term or demand.

- Environmental commodities C_{env}: this constraint is not defined for environmental commodities.
- Suppy intermittent commodities C_{sup}: this constraint is not defined for supply intermittent commodities.
- Stock commodities C_{st}: For stock commodities, the possible imbalance of the commodity must be supplied by the stock commodity purchases. In other words, commodity balance $\mathrm{CB}(v, c, t)$ subtracted from the variable stock commodity source term ρ_{vct} must be greater than or equal to 0 to satisfy this constraint. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{st}}, t \in T_m\colon\ \ -\mathrm{CB}(v, c, t) + \rho_{vct} \geq 0$$

- Sell commodities C_{sell}: For sell commodities, the possible imbalance of the commodity must be supplied by the sell commodity trades. In other words, commodity balance $\mathrm{CB}(v, c, t)$ subtracted from minus the variable sell commodity source term ϱ_{vct} must be greater than or equal to 0 to satisfy this constraint. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{sell}}, t \in T_m\colon\ \ -\mathrm{CB}(v, c, t) - \varrho_{vct} \geq 0$$

- Buy commodities C_{buy}: For buy commodities, the possible imbalance of the commodity must be supplied by the buy commodity purchases. In other words, commodity balance $\mathrm{CB}(v, c, t)$ subtracted from the variable buy commodity source term ψ_{vct} must be greater than or equal to 0 to satisfy this constraint. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{buy}}, t \in T_m\colon\ \ -\mathrm{CB}(v, c, t) + \psi_{vct} \geq 0$$

- Demand commodities C_{dem}: For demand commodities, the possible imbalance of the commodity must supply the demand d_{vct} of demand commodities $c \in C_{\text{dem}}$. In other words, the parameter demand for commodity subtracted d_{vct} from the minus commodity balance $-\mathrm{CB}(v, c, t)$ must be greater than or equal to 0 to satisfy this constraint. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{dem}}, t \in T_m\colon\ \ -\mathrm{CB}(v, c, t) - d_{vct} \geq 0$$

- Demand Side Management commodities and sites: For any combination of commodity and site for which demand side management is defined, the upshift is substracted and the downshift added to the negative commodity balance $-\mathrm{CB}(v, c, t)$.

$$\forall (v,c) in D_{vc}, t \in T_m\colon\ \ -\mathrm{CB}(v, c, t) - \delta_{vct}^{\text{up}}` + \sum_{tt \in D_{vct,tt}^{\text{down}}} \delta_{vct,tt}^{\text{down}}` \geq 0$$

In script `urbs.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')
```

Stock Per Step Rule: The constraint stock per step rule applies only for commodities of type "Stock" ($c \in C_{\text{st}}$). This constraint limits the amount of stock commodity $c \in C_{\text{st}}$, that can be used by the energy system in the site v at the timestep t. The limited amount is defined by the parameter maximum stock supply limit per time step $\bar{l}_{vc}$. To satisfy this constraint, the value of the variable stock commodity source term ρ_{vct} must be less than or equal to the value of the parameter maximum stock supply limit per time step $\bar{l}_{vc}$. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{st}}, t \in T_m\colon\ \rho_{vct} \leq \bar{l}_{vc}$$

In script `urbs.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')
```

Total Stock Rule: The constraint total stock rule applies only for commodities of type "Stock" ($c \in C_{\text{st}}$). This constraint limits the amount of stock commodity $c \in C_{\text{st}}$, that can be used annually by the energy system in the site v. The limited amount is defined by the parameter maximum annual stock supply limit per vertex $\overline{L}_{vc}$. To satisfy this constraint, the annual usage of stock commodity must be less than or equal to the value of the parameter stock supply limit per vertex $\overline{L}_{vc}$. The annual usage of stock commodity is calculated by the sum of the products of the parameter weight w, the parameter timestep duration Δt and the parameter stock commodity source term ρ_{vct} for every timestep $t \in T_m$. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{st}}\colon\ w \sum_{t \in T_m} \Delta t \, \rho_{vct} \leq \overline{L}_{vc}$$

In script `urbs.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')
```

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 at the timestep t. The limited amount is defined by the parameter maximum sell supply limit per time step $\overline{g}_{vc}$. To satisfy this constraint, the value of the variable sell commodity source term ϱ_{vct} must be less than or equal to the value of the parameter maximum sell supply limit per time step $\overline{g}_{vc}$. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{sell}}, t \in T_m\colon\ \varrho_{vct} \leq \overline{g}_{vc}$$

In script `urbs.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')
```

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. The limited amount is defined by the parameter maximum annual sell supply limit per vertex $\overline{G}_{vc}$. To satisfy this constraint, the annual usage of sell commodity must be less than or equal to the value of the parameter sell supply limit per vertex $\overline{G}_{vc}$. The annual usage of sell commodity is calculated by the sum of the products of the parameter weight w, the parameter timestep duration Δt and the parameter sell commodity source term ϱ_{vct} for every timestep $t \in T_m$. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_\text{sell}\colon\ w \sum_{t\in T_m} \Delta t\, \varrho_{vct} \leq \overline{G}_{vc}$$

In script `urbs.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')
```

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 at the timestep t. The limited amount is defined by the parameter maximum buy supply limit per time step $\overline{b}_{vc}$. To satisfy this constraint, the value of the variable buy commodity source term ψ_{vct} must be less than or equal to the value of the parameter maximum buy supply limit per time step $\overline{b}_{vc}$. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_\text{buy}, t \in T_m\colon\ \psi_{vct} \leq \overline{b}_{vc}$$

In script `urbs.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')
```

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. The limited amount is defined by the parameter maximum annual buy supply limit per vertex $\overline{B}_{vc}$. 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, the parameter timestep duration Δt and the parameter buy commodity source term ψ_{vct} for every timestep $t \in T_m$. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_\text{buy}\colon\ w \sum_{t\in T_m} \Delta t\, \psi_{vct} \leq \overline{B}_{vc}$$

In script `urbs.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')
```

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 at the timestep t.

The limited amount is defined by the parameter maximum environmental output per time step $\overline{m}_{vc}$. 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 $\overline{m}_{vc}$. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{env}}, t \in T_m\colon \quad -\operatorname{CB}(v,c,t) \leq \overline{m}_{vc}$$

In script `urbs.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')
```

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. The limited amount is defined by the parameter maximum annual environmental output limit per vertex $\overline{M}_{vc}$. 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, the parameter timestep duration Δt and the negative value of commodity balance function, for every timestep $t \in T_m$. In mathematical notation this is expressed as:

$$\forall v \in V, c \in C_{\text{env}}\colon \quad -w \sum_{t \in T_m} \Delta t \operatorname{CB}(v,c,t) \leq \overline{M}_{vc}$$

In script `urbs.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')
```

In script `urbs.py` the constraint total environmental output rule is defined and calculated by the following code fragment:

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 upshift and downshift. An upshift δ_{vct}^{up} in site v of commodity c in time step t can be compensated during a certain time interval $[t - y_{vc}, t + y_{vc}]$ by multiple downshifts $\delta_{vct,tt}^{\text{down}}$. Depending on the efficiency e_{vc}, less downshifts have to be made. This is given by:

$$\forall (v,c) \in D_{vc}, t \in T\colon \quad \delta_{vct}^{\text{up}} e_{vc} = \sum_{tt=t-y_{vc}}^{t+y_{vc}} \delta_{vct,tt}^{\text{down}}$$

The definition of the constraint and its corresponding rule is defined by the following code:

```
m.def_dsm_variables = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=def_dsm_variables_rule,
    doc='DSMup == DSMdo * efficiency factor n')
```

DSM Upward Rule: The DSM upshift δ_{vct}^{up} in site v of commodity c in time step t is limited by the maximal upshift capacity $\overline{K}_{vc}^{\text{up}}$. In mathematical terms, this is written as:

$$\forall (v,c) \in D_{vc}, t \in T\colon\ \delta_{vct}^{\text{up}} \leq \overline{K}_{vc}^{\text{up}}$$

The definition of the constraint and its corresponding rule is defined by the following code:

```
m.res_dsm_upward = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=res_dsm_upward_rule,
    doc='DSMup <= Cup (threshold capacity of DSMup)')
```

DSM Downward Rule: The DSM downshift δ_{vct}^{up} in site v of commodity c in time step t is limited by the maximal upshift capacity $\overline{K}_{vc}^{\text{up}}$. In mathematical terms, this is written as:

$$\forall (v,c) \in D_{vc}, tt \in T\colon\ \sum_{t=tt-y}^{tt+y} \delta_{vct,tt}^{\text{down}} \leq \overline{K}_{vc}^{\text{down}}$$

The definition of the constraint and its corresponding rule is defined by the following code:

```
m.res_dsm_downward = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=res_dsm_downward_rule,
    doc='DSMdo <= Cdo (threshold capacity of DSMdo)')
```

DSM Maximum Rule: The DSM maximum rule limits the shift of one DSM unit in site v of commodity c in time step t. In mathematical terms, this is written as:

$$\forall (v,c) \in D_{vc}, tt \in T\colon\ \delta_{vct}^{\text{up}} + \sum_{t=tt-y}^{tt+y} \delta_{vct,tt}^{\text{down}} \leq \max \left\{ \overline{K}_{vc}^{\text{up}}, \overline{K}_{vc}^{\text{down}} \right\}$$

The definition of the constraint and its corresponding rule is defined by the following code:

```
m.res_dsm_maximum = pyomo.Constraint(
    m.tm, m.dsm_site_tuples,
    rule=res_dsm_maximum_rule,
    doc='DSMup + DSMdo <= max(Cup,Cdo)')
```

DSM Recovery Rule: The DSM recovery rule limits the upshift in site v of commodity c during a set recovery period o_{vc}. In mathematical terms, this is written as:

$$\forall (v,c) \in D_{vc}, t \in T\colon\ \sum_{tt=t}^{t+o_{vc}-1} \delta_{vctt}^{\text{up}} \leq \overline{K}_{vc}^{\text{up}} y$$

The definition of the constraint and its corresponding rule is defined by the following code:

```
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')
```

Global Environmental Constraint

Global CO2 Limit Rule: The constraint global CO2 limit rule applies to the whole energy system, that is to say it applies to every site and timestep in general. This constraints restricts the energy model from releasing more environmental commodities, namely CO2 to environment than allowed. The constraint states that the sum of released environmental commodities for every site v and every timestep t must be less than or equal to the parameter maximum global annual CO2 emission limit $\overline{L}_{CO_2}$, where the amount of released enviromental commodites in a single site v at a single timestep t is calculated by the product of commodity balance of enviromental commodities $\mathrm{CB}(v, CO_2, t)$ and the parameter weight w. This constraint is skipped if the value of the parameter $\overline{L}_{CO_2}$ is set to `inf`. In mathematical notation this constraint is expressed as:

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

In script `urbs.py` the constraint global CO2 limit rule is defined and calculated by the following code fragment:

Process Constraints

Process Capacity Rule: The constraint process capacity rule defines the variable total process capacity κ_{vp}. 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 $\hat{\kappa}_{vp}$. In mathematical notation this is expressed as:

$$\forall v \in V, p \in P\colon\ \kappa_{vp} = K_{vp} + \hat{\kappa}_{vp}$$

In script `urbs.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')
```

Process Input Rule: The constraint process input rule defines the variable process input commodity flow $\epsilon_{vcpt}^\text{in}$. The variable process input commodity flow is defined by the constraint as the product of the variable process throughput τ_{vpt} and the parameter process input ratio r_{pc}^in. In mathematical notation this is expressed as:

$$\forall v \in V, p \in P, t \in T_m\colon\ \epsilon_{vcpt}^\text{in} = \tau_{vpt} r_{pc}^\text{in}$$

In script `urbs.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')
```

Process Output Rule: The constraint process output rule defines the variable process output commodity flow $\epsilon_{vcpt}^\text{out}$. The variable process output commodity flow is defined by the constraint as the product of the variable process throughput τ_{vpt} and the parameter process output ratio r_{pc}^{out}. In mathematical notation this is expressed as:

$$\forall v \in V, p \in P, t \in T_m\colon\ \epsilon_{vpct}^{\text{out}} = \tau_{vpt} r_{pc}^{\text{out}}$$

In script `urbs.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,
    rule=def_process_output_rule,
    doc='process output = process throughput * output ratio')
```

Intermittent Supply Rule: The constraint intermittent supply rule defines the variable process input commodity flow $\epsilon_{vcpt}^{\text{in}}$ 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_{vp} and the parameter intermittent supply capacity factor s_{vct}. In mathematical notation this is expressed as:

$$\forall v \in V, p \in P, c \in C_{\text{sup}}, t \in T_m\colon\ \epsilon_{vpct}^{\text{in}} = \kappa_{vp} s_{vct}$$

In script `urbs.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')
```

Process Throughput By Capacity Rule: The constraint process throughput by capacity rule limits the variable process throughput τ_{vpt}. 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_{vp}. In mathematical notation this is expressed as:

$$\forall v \in V, p \in P, t \in T_m\colon\ \tau_{vpt} \leq \kappa_{vp}$$

In script `urbs.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')
```

Process Throughput Gradient Rule: The constraint process throughput gradient rule limits the process power gradient $\left|\tau_{vpt} - \tau_{vp(t-1)}\right|$. This constraint prevents processes from exceeding their maximal possible change in activity from one time step to the next. The constraint states that absolute power gradient must be less than or equal to the maximal power gradient $\overline{PG}_{vp}$ parameter (scaled to capacity and by time step duration). In mathematical notation this is expressed as:

$$\forall v \in V, p \in P, t \in T_m\colon\ \left|\tau_{vpt} - \tau_{vp(t-1)}\right| \leq \kappa_{vp} \overline{PG}_{vp} \Delta t$$

In script `urbs.py` the constraint process throughput gradient rule is defined and calculated by the following code fragment:

```
m.res_process_throughput_gradient = pyomo.Constraint(
    m.tm, m.pro_tuples,
    rule=res_process_throughput_gradient_rule,
    doc='process throughput gradient <= maximal gradient')
```

Process Capacity Limit Rule: The constraint process capacity limit rule limits the variable total process capacity κ_{vp}. This constraint restricts a process p in a site v 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_{vp} must be greater than or equal to the parameter process capacity lower bound $\underline{K}_{vp}$ and less than or equal to the parameter process capacity upper bound $\overline{K}_{vp}$. In mathematical notation this is expressed as:

$$\forall v \in V, p \in P\colon\ \underline{K}_{vp} \leq \kappa_{vp} \leq \overline{K}_{vp}$$

In script `urbs.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')
```

Sell Buy Symmetry Rule: The constraint sell buy symmetry rule defines the total process capacity κ_{vp} of a process p in a site v 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_{vp} 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.

In script `urbs.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')
```

Partial & Startup Process Constraints

It is important to understand that this partial load formulation can only work if its accompanied by a sensible value for both the minimum partial load fraction $\underline{P}_{vp}$ and the startup cost k_{vp}^{startup}. Otherwise, the optimal solution yields identical operation and performance like a regular, fully proportional process with constant/flat input ratios.

Throughput by Online Capacity Min Rule

The new variable *online capacity* forces the process throughput to always stay above its value times the minium partial load fraction. But note that there is **no** constraint that stops ω_{vpt} from assuming arbitrarily small values. This is only softly prohibited by the startup cost term, which acts as kind of a soft friction term that punishes too dynamic of an operation strategy.

$$\forall t \in T_{\text{m}}, (v,p) \in P_v^{\text{partial}}\colon\ \tau_{vpt} \geq \omega_{vpt} \underline{P}_{vp}$$

And here as code:

```
m.res_throughput_by_online_capacity_min = pyomo.Constraint(
    m.tm, m.pro_partial_tuples,
    rule=res_throughput_by_online_capacity_min_rule,
    doc='cap_online * min-fraction <= tau_pro')
```

Throughput by Online Capacity Max Rule

On the other side, the *online capacity* is an upper cap on the process throughput.

$$\forall t \in T_\text{m}, (v,p) \in P_v^\text{partial}\colon \ \tau_{vpt} \leq \omega_{vpt}$$

And the code:

```
m.res_throughput_by_online_capacity_max = pyomo.Constraint(
    m.tm, m.pro_partial_tuples,
    rule=res_throughput_by_online_capacity_max_rule,
    doc='tau_pro <= cap_online')
```

Partial Process Input Rule: In energy system modelling, the simplest way to represent an energy conversion process is to assume a linear input-output relationship with a flat efficiency parameter η:

$$\epsilon_{out} = \epsilon_{in} \cdot \eta$$

Which means there is only one efficiency η during the whole process, i.e. it remains constant during the electricity production. But in fact, most of the powerplants do not operate at a certain efficiency and the operation load varies along time. Therefore the regular single efficiency η will be replaced by a set of input ratios (r^in) and output ratios (r^out) in urbs. And both ratios relate to the process activity τ:

$$\epsilon_{pct}^\text{in} = \tau_{pt} r_{pc}^\text{in}$$
$$\epsilon_{pct}^\text{out} = \tau_{pt} r_{pc}^\text{out}$$

In order to simplify the mathematical calculation, the output ratios are set to 1 so that the process output ($\epsilon_{pct}^\text{out}$) is equal to the process throughput (τ). Then, the process efficiency η can be represented as follows:

$$\eta = \frac{\epsilon_{pct}^\text{out}}{\epsilon_{pct}^\text{in}} = \frac{\tau}{\epsilon_{pct}^\text{in}}$$

Assume now a process, it has a lower input ratio $\underline{r}_{pc}^\text{in}$, a upper input ratio r_{pc}^in, the process minimum part load fraction $\underline{P}_{vp}$ and the corresponding start-up costs. The τ will be bounded by $\underline{P}_{vp}$ and the online capacity ω_{vpt}, which means the throughput can only vary between $\underline{P}_{vp} \cdot \omega_{vpt}$ and ω_{vpt}. When all the start-up costs are equal to zero, the relation between the process input and the process throughout is nothing else but a straight line across the original point, which exists almost only theoretically. Practically, every powerplant has a start-up cost, which has a big influence on the effeiciency of the process.

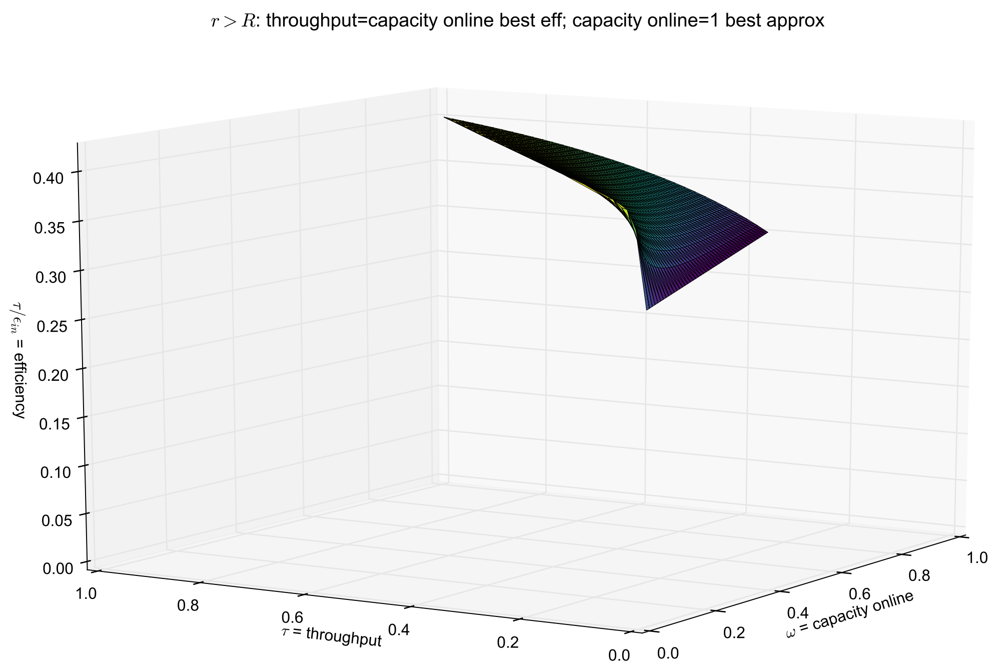
To research the influence of the start-up costs, a continouous start-up variable $\chi_{pt} \in [0, \kappa_p]$ is introduced and defines as follows:

$$\tau_{pt} \leq \omega_{pt}$$
$$\chi_{pt} \geq \omega_{pt} - \omega_{p(t-1)}$$
$$\zeta_\text{var} \mathrel{+}= \sum_{t\in T} \sum_{p\in P} k_p^\text{startup} \chi_{pt}$$

Where the ω_{pt} is also a new introduced variable, represents the start-up capacity (or the idle consumption). With these two variables, the urbs can detect the energy consumption of a process at the starting point and put a start-up costs on it to obtain the variable costs.

$$\forall t \in T_\text{m}, (v,p,c) \in C_{vp}^\text{in,partial}\colon\ \epsilon_{vpct}^\text{in} = \omega_{vpt} \cdot \frac{\underline{r}_{pc}^\text{in} - r_{pc}^\text{in}}{1 - \underline{P}_{vp}} \cdot \underline{P}_{vp} + \tau_{vpt} \cdot \frac{r_{pc}^\text{in} - \underline{P}_{vp}\underline{r}_{pc}^\text{in}}{1 - \underline{P}_{vp}}$$

As it is not immediately clear what this expression accomplishes, here is visual example. It plots the value off the expression $\tau_{vpt}/\epsilon_{vpct}^\text{in}$ for a process with $\underline{P}_{vp} = 0.35$, $\underline{r}_{pc}^\text{in} = 3.33$ and $r_{pc}^\text{in} = 2.5$ and a hypothetical capacity of $1MW$. When operating at its maximum, it yields an input efficiecny of 40%, whereas in partial load this drops to 30%.



More discussion and a visualisation of the reverse case (partial load more efficient than full load operation) is shown in a dedicated IPython notebook.

```
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_online * min_fraction * (r - R) / (1 - min_
↪fraction)'
                    '+ tau_pro * (R - min_fraction * r) / (1 - min_
↪fraction)')
```

Online Capacity By Process Capacity Rule limits the value of the online capacity ω_{vpt} by the total installed process capacity κ_{vp}:

$$\forall (v,p) \in P_v^\text{partial}, t \in T_\text{m}\colon\ \omega_{vpt} \leq \kappa_{vp}$$

```
m.res_cap_online_by_cap_pro = pyomo.Constraint(
    m.tm, m.pro_partial_tuples,
    rule=res_cap_online_by_cap_pro_rule,
    doc='online capacity <= process capacity')
```

Startup Capacity Rule determines the value of the startup capacity indicator variable ϕ_{vpt}, by limiting its value to at least the positive difference of subsequent online capacity states ω_{vpt} and $\omega_{vp(t-1)}$. In other words: whenever the onlince capacity increases, startup capacity ϕ_{vpt} assumes a non-zero value.

$$\forall (v,p) \in P_v^{\text{partial}}, t \in T_\text{m}\colon\ \phi_{vpt} \geq \omega_{vpt} - \omega_{vp(t-1)}$$

Code declaration and definition:

```
m.def_startup_capacity = pyomo.Constraint(
    m.tm, m.pro_partial_tuples,
    rule=def_startup_capacity_rule,
    doc='startup_capacity[t] >= cap_online[t] - cap_online[t-1]')
```

Transmission Constraints

Transmission Capacity Rule: The constraint transmission capacity rule defines the variable total transmission capacity κ_{af}. The variable total transmission capacity is defined by the constraint as the sum of the variable transmission capacity installed K_{af} and the variable new transmission capacity $\hat{\kappa}_{af}$. In mathematical notation this is expressed as:

$$\forall a \in A, f \in F\colon\ \kappa_{af} = K_{af} + \hat{\kappa}_{af}$$

In script `urbs.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')
```

Transmission Output Rule: The constraint transmission output rule defines the variable transmission power flow (output) π_{aft}^out. The variable transmission power flow (output) is defined by the constraint as the product of the variable transmission power flow (input) π_{aft}^in and the parameter transmission efficiency e_{af}. In mathematical notation this is expressed as:

$$\forall a \in A, f \in F, t \in T_m\colon\ \pi_{aft}^\text{out} = \pi_{aft}^\text{in} e_{af}$$

In script `urbs.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')
```

Transmission Input By Capacity Rule: The constraint transmission input by capacity rule limits the variable transmission power flow (input) π_{aft}^in. This constraint prevents transmissions from exceeding their possible power input capacity. The constraint states that the variable transmission power flow (input) π_{aft}^in must be less than or equal to the variable total transmission capacity κ_{af}. In mathematical notation this is expressed as:

$$\forall a \in A, f \in F, t \in T_m\colon\ \pi_{aft}^\text{in} \leq \kappa_{af}$$

In script `urbs.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')
```

Transmission Capacity Limit Rule: The constraint transmission capacity limit rule limits the variable total transmission capacity κ_{af}. This constraint restricts a transmission f through an arc a 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_{af} must be greater than or equal to the parameter transmission capacity lower bound $\underline{K}_{af}$ and less than or equal to the parameter transmission capacity upper bound $\overline{K}_{af}$. In mathematical notation this is expressed as:

$$\forall a \in A, f \in F\colon\ \underline{K}_{af} \leq \kappa_{af} \leq \overline{K}_{af}$$

In script `urbs.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')
```

Transmission Symmetry Rule: The constraint transmission symmetry rule defines the power output capacities of incoming and outgoing arcs a, a' of a transmission f. The constraint states that the power output capacities κ_{af} of the incoming arc a and the complementary outgoing arc a' between two sites must be equal. In mathematical notation this is expressed as:

$$\forall a \in A, f \in F\colon\ \kappa_{af} = \kappa_{a'f}$$

In script `urbs.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')
```

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 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 $t-1$. Here ingoing energy is given by the product of the variable input storage power flow $\epsilon_{vst}^{\text{in}}$, the parameter timestep duration Δt and the parameter storage efficiency during charge e_{vs}^{in}. Outgoing energy is given by the product of the variable output storage power flow $\epsilon_{vst}^{\text{out}}$ and the parameter timestep duration Δt divided by the parameter storage efficiency during discharge e_{vs}^{out}. In mathematical notation this is expressed as:

$$\forall v \in V, \forall s \in S, t \in T_{\text{m}}\colon\ \epsilon_{vst}^{\text{con}} = \epsilon_{vs(t-1)}^{\text{con}} + \epsilon_{vst}^{\text{in}} \cdot e_{vs}^{\text{in}} - \epsilon_{vst}^{\text{out}} / e_{vs}^{\text{out}}$$

In script `urbs.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] = storage[t-1] + input - output')
```

Storage Power Rule: The constraint storage power rule defines the variable total storage power κ_{vs}^{p}. The variable total storage power is defined by the constraint as the sum of the parameter storage power installed K_{vs}^{p} and the variable new storage power $\hat{\kappa}_{vs}^{\text{p}}$. In mathematical notation this is expressed as:

$$\forall v \in V, s \in S\colon\ \kappa_{vs}^{\text{p}} = K_{vs}^{\text{p}} + \hat{\kappa}_{vs}^{\text{p}}$$

In script `urbs.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')
```

Storage Capacity Rule: The constraint storage capacity rule defines the variable total storage size κ_{vs}^{c}. The variable total storage size is defined by the constraint as the sum of the parameter storage content installed K_{vs}^{c} and the variable new storage size $\hat{\kappa}_{vs}^{\text{c}}$. In mathematical notation this is expressed as:

$$\forall v \in V, s \in S\colon\ \kappa_{vs}^{\text{c}} = K_{vs}^{\text{c}} + \hat{\kappa}_{vs}^{\text{c}}$$

In script `urbs.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')
```

Storage Input By Power Rule: The constraint storage input by power rule limits the variable storage input power flow $\epsilon_{vst}^{\text{in}}$. This constraint restricts a storage s in a site v at a timestep t from having more input power than the storage power capacity. The constraint states that the variable $\epsilon_{vst}^{\text{in}}$ must be less than or equal to the variable total storage power κ_{vs}^{p}. In mathematical notation this is expressed as:

$$\forall v \in V, s \in S, t \in T_m\colon\ \epsilon_{vst}^{\text{in}} \leq \kappa_{vs}^{\text{p}}$$

In script `urbs.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')
```

Storage Output By Power Rule: The constraint storage output by power rule limits the variable storage output power flow $\epsilon_{vst}^{\text{out}}$. This constraint restricts a storage s in a site v at a timestep t from having more output power than the storage power capacity. The constraint states that the variable $\epsilon_{vst}^{\text{out}}$ must be less than or equal to the variable total storage power κ_{vs}^{p}. In mathematical notation this is expressed as:

$$\forall v \in V, s \in S, t \in T\colon\ \epsilon_{vst}^{\text{out}} \leq \kappa_{vs}^{\text{p}}$$

In script `urbs.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')
```

Storage State By Capacity Rule: The constraint storage state by capacity rule limits the variable storage energy content $\epsilon_{vst}^\text{con}$. This constraint restricts a storage s in a site v at a timestep t from having more storage content than the storage content capacity. The constraint states that the variable $\epsilon_{vst}^\text{con}$ must be less than or equal to the variable total storage size κ_{vs}^c. In mathematical notation this is expressed as:

$$\forall v\in V, s\in S, t\in T\colon\ \epsilon_{vst}^\text{con} \leq \kappa_{vs}^\text{c}$$

In script `urbs.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')
```

Storage Power Limit Rule: The constraint storage power limit rule limits the variable total storage power κ_{vs}^p. This contraint restricts a storage s in a site v 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 κ_{vs}^p must be greater than or equal to the parameter storage power lower bound $\underline{K}_{vs}^\text{p}$ and less than or equal to the parameter storage power upper bound $\overline{K}_{vs}^\text{p}$. In mathematical notation this is expressed as:

$$\forall v\in V, s\in S\colon\ \underline{K}_{vs}^\text{p} \leq \kappa_{vs}^\text{p} \leq \overline{K}_{vs}^\text{p}$$

In script `urbs.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')
```

Storage Capacity Limit Rule: The constraint storage capacity limit rule limits the variable total storage size κ_{vs}^c. This contraint restricts a storage s in a site v 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 κ_{vs}^c must be greater than or equal to the parameter storage content lower bound $\underline{K}_{vs}^\text{c}$ and less than or equal to the parameter storage content upper bound $\overline{K}_{vs}^\text{c}$. In mathematical notation this is expressed as:

$$\forall v\in V, s\in S\colon\ \underline{K}_{vs}^\text{c} \leq \kappa_{vs}^\text{c} \leq \overline{K}_{vs}^\text{c}$$

In script `urbs.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')
```

Initial And Final Storage State Rule: The constraint initial and final storage state rule defines and restricts the variable storage energy content $\epsilon_{vst}^\text{con}$ of a storage s in a site v at the initial timestep t_1 and at the final timestep t_N.

Initial Storage: Initial storage represents how much energy is installed in a storage at the beginning of the simulation. The variable storage energy content $\epsilon_{vst}^\text{con}$ at the initial timestep t_1 is defined by this constraint. The constraint states that the variable $\epsilon_{vst_1}^\text{con}$ must be equal to the product of the parameters storage content installed K_{vs}^c and initial and final state of charge I_{vs}. In mathematical notation this is expressed as:

$$\forall v \in V, s \in S\colon\ \epsilon_{vst_1}^\text{con} = \kappa_{vs}^\text{c} I_{vs}$$

Final Storage: Final storage represents how much energy is installed in a storage at the end of the simulation. The variable storage energy content $\epsilon_{vst}^\text{con}$ at the final timestep t_N is restricted by this constraint. The constraint states that the variable $\epsilon_{vst_N}^\text{con}$ must be greater than or equal to the product of the parameters storage content installed K_{vs}^c and initial and final state of charge I_{vs}. In mathematical notation this is expressed as:

$$\forall v \in V, s \in S\colon\ \epsilon_{vst_N}^\text{con} \geq \kappa_{vs}^\text{c} I_{vs}$$

In script `urbs.py` the constraint initial and final storage state rule is defined and calculated by the following code fragment:

```
m.res_initial_and_final_storage_state = pyomo.Constraint(
    m.t, m.sto_tuples,
    rule=res_initial_and_final_storage_state_rule,
    doc='storage content initial == and final >= storage.init * capacity')
```

1.2.4 Buy-Sell Documentation

This documentation explains the buy-sell-price feature of urbs. With it one can model time variant electricity prices from energy exchanges.

Introduction

The prices are independent of the amount of electricity purchased and fed in as there is no feedback. The size of the modelled market has to be considered small relative to the surrounding market. To use this feature your excel input file needs an additional **Buy-Sell-Price** sheet with the columns `t` containing the timesteps and the columns `Elec buy` and `Elec sell` containing the buy and sell prices by default in hourly € per MWh. In the **Commodity** sheet the price for `Elec` at a `Site` has to be changed from a number to a string `Buy` or `Sell` or a multiple of it for example `1,25xBuy`. For a more detailed description of the implementation have a look at the Mathematical Documentation.

Exemplification

This section contains prototypical scenarios illustrating the system behaviour with time variant prices. Electricity can be moved *locally* with transmission losses and *temporally* with storage losses.

Fix Capacities - Fix Prices

All process, transmission and storage capacities and prices are predetermined and constant.

When is electricity purchased?

- if it is *necessary* that is the demand is greater than the total output capacity it is bought at every price
- if it is *profitable* that is if the buy price is lesser than the variable costs of the most expensive needed process

When is electricity fed-in?

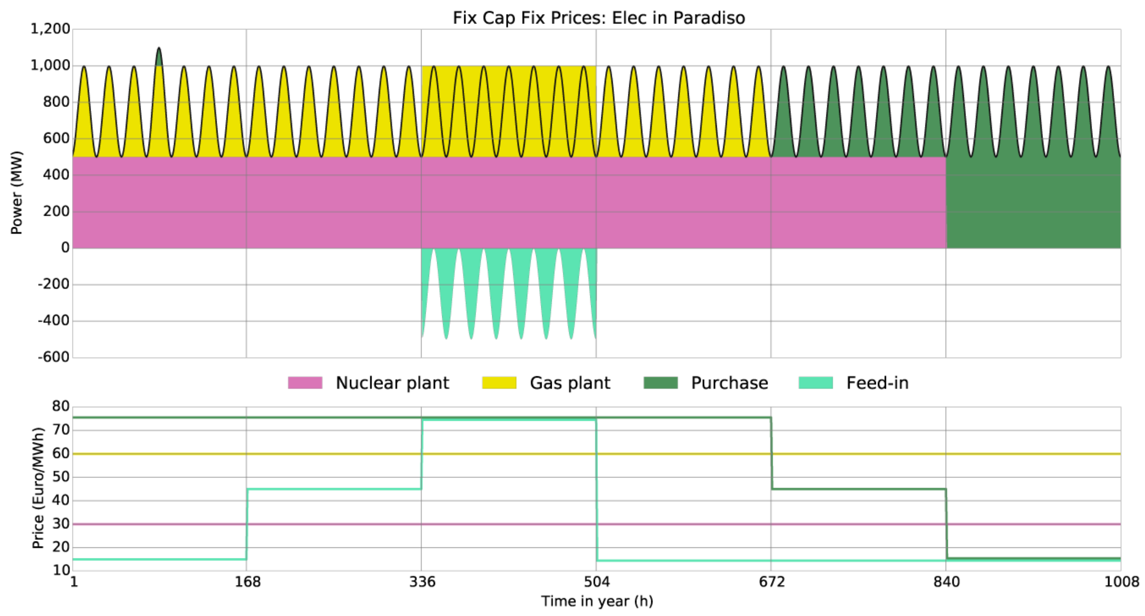
- if it is *possible* **and** *profitable* that is if the demand is lesser than the total output capacity **and** the sell price greater than the cheapest currently not needed process

The following scenario illustrates the energy balance of the island Paradiso. It has a demand of 500-1000 MW that is supplied by a 1500 MW nuclear plant, a 1000 MW gas plant and a 1000 MW transmission cable, that connects the island grid with the continental grid. Both capacities and prices are fix.

Table 14: Scenario Fix Cap Fix Prices

Process	eff	inst-cap	inst-cap-out	fuel-cost	var-cost	total-var-cost
Nuclear plant	0.33	1500	500	5	5	10
Gas plant	0.50	1000	500	25	5	30
Purchase	1.00	1000	1000	**15/45/75**	0	15/45/75
Feed-in	1.00	1000	1000	**15/45/75**	0	15/45/75

The modelled timespan is 6 weeks with different fix prices each. In week 1 on the fourth day energy is purchased, because it is neccessary to cover the demand. In week 2 the sell price is higher than the variable costs of the nuclear plant, but lower than the variable costs of the cheapest not needed power plant: the gas plant. In week 3 the sell price excels even those costs making the production and selling of additional energy profitable. In week 4 buy prices are too high for purchase and sell prices to low for feed-in. In week 5 buy prices have dropped enough for purchased energy to replace energy produced by the gas plant. In week 6 they further dropped enough to even replace energy produced by the nuclear plant.



Fix Capacities - Variable Prices

All process, transmission and storage capacities are predetermined and constant, prices are varying over the modelled timespan.

When is electricity purchased?

- if it is *necessary* that is the demand is greater than the total output capacity it is bought at every price
- if it is *profitable* that is if the buy price is lesser than the current variable costs of the most expensive needed process *or* including storage costs lesser than future variable costs of the most expensive needed process

When is electricity fed-in?

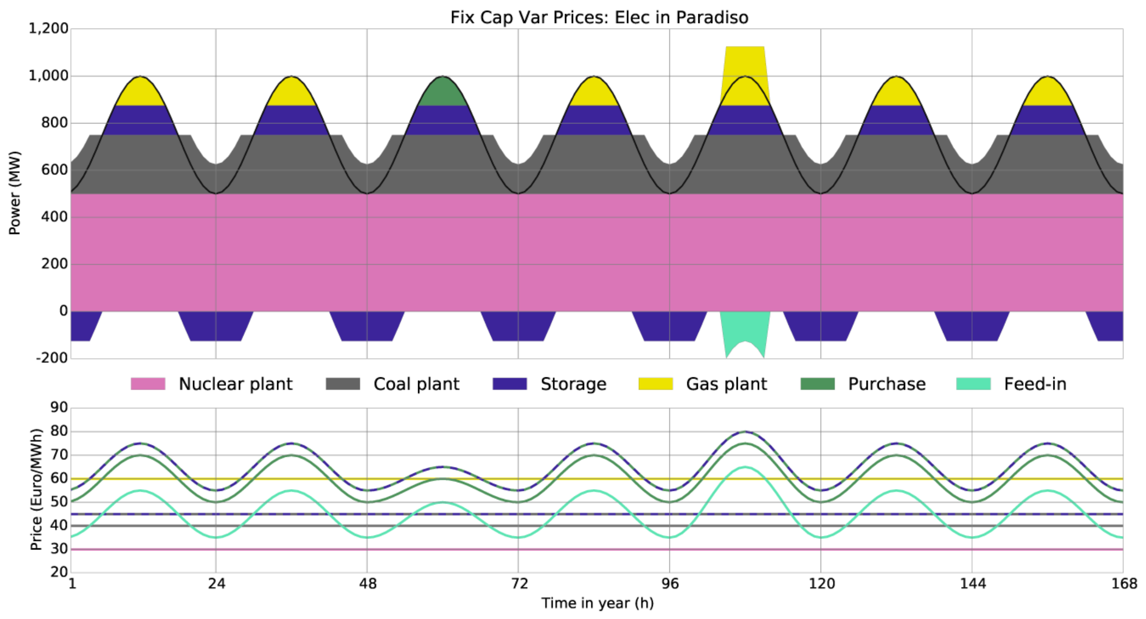
- if it is *possible* **and** *profitable* that is if the demand is lesser than the total output capacity **and** the sell price greater than the cheapest currently not needed process

For the second scenario half of the gas plant is replaced by a coal plant. Additionally there is a new power limited energy storage with variable storage costs of 5 €/MWh. The load curve stays the same. Capacities are fix and prices are varying.

Table 15: Scenario Fix Cap Var Prices

Process	eff	inst-cap	inst-cap-out	fuel-cost	var-cost	total-var-cost
Nuclear plant	0.33	1500	500	5	5	10
Coal Plant	0.40	625	250	11	5	16
Gas plant	0.50	500	250	25	5	30
Storage	1.00	125	125		2.5	5
Purchase	1.00	1000	1000	**50-75**	0	50-75
Feed-in	1.00	1000	1000	**35-65**	0	35-65

The modelled timespan is 7 days. The buy price varies around the variable costs of the gas plant. But except for day 3 purchase is only a profitable substitute for energy from the gas plant at timesteps it is not needed. The sell price varies around the variable costs of the coal plant. But similar to the buy price except for day 5 it only allows production of energy for selling at timesteps it required to cover the demand instead. Producing and storing energy from the coal plant at timesteps with a low demand limited only by the storage power capacity is profitable, because it has total variable costs of 45 €/MWh and substitutes ebergy from the gas plant costing 60 €/MWh. At day 5 at noon the sell price exceeds the purchase price 12 hours before by 15 €/MWh. Even discounting storage costs of 5 €/MWh it would allow infinite arbitrage. But since the storage capacities are limited the opportunity costs of 15 €/MWh of substituting energy from the gas plant are higher than the 10 €/MWh profit margin it is not done.



Note: For trial e.g. of the result of greater storage capacities this `paradiso_2.xlsx` is the input file used for this scenario.

Variable Capacities - Variable Prices

All process, transmission and storage capacities are variable and determined at optimal total cost, prices are varying over the modelled timespan.

When is electricity purchased?

- if it is *necessary* that is the demand is greater than the total output capacity it is bought at every price
- if it is *profitable* that is if the buy price is lesser than the current variable costs of the most expensive needed process *or* including storage costs lesser than future variable costs of the most expensive needed process *or* it reduces the peak load allowing the capacity investments to be reduced in a way that overcompensates the additional costs in summary

When is electricity fed-in?

- if it is *possible* **and** *profitable* that is if the demand is lesser than the total output capacity **and** the sell price greater than the cheapest currently not needed process *and* does not prevent a total costs decrease by reduction of the capacity investments

The next scenario is very similar to the previous one, only that this time all capacities are initially 0 and investment in new capacities is done in a cost optimal way. The ascencing order of variable prices is still nuclear plant - coal plant - gas plan. The ascending order of fix costs, the sum of annual fix costs `fix-cost` and annualized depreciations calculated from the investment costs `inv-cost`, weighted average cost of capital `wacc` and economic life time `depreciation` is the opposite: gas plant - coal plant - nuclear plant.

Table 16: Scenario Var Cap Var Prices (1)

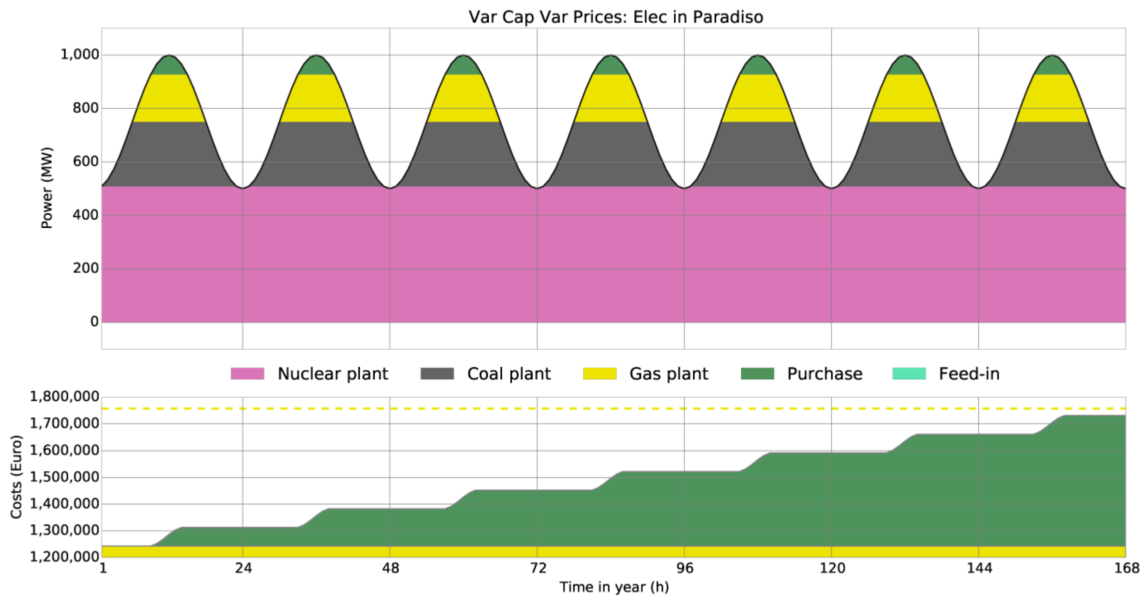
Process	eff	**inst-cap**	inst-cap-out	fuel-cost	var-cost	total-var-cost
Nuclear plant	0.33	0	0	5	5	10
Coal Plant	0.40	0	0	11	5	16
Gas plant	0.50	0	0	25	5	**30**
Storage	1.00	0	0		2.5	5
Purchase	1.00	0	0	150-250	0	**150-250**
Feed-in	1.00	0	0	30-50	0	30-50

This scenario should demonstrate a typical composition of power plants. This is the result of each power plant being cost optimal for a certain range of full load hours per year leading nuclear energy to cover the base load and gas energy to cover the peak load. It should also demonstrate, why the purchase of energy that at the moment exceeds variable costs of power plants can be economically worthwhile as it reduces peak loads and decreases overall costs.

Table 17: Scenario Var Cap Var Prices (2)

Process	fix-cost	inv-costs	wacc	depreciation	anf	annuity	total-fix-cost
Gas plant	2000	2250000	0.07	30	0.08	181319	**183319**
Purchase	0	0	0.07			0	**0**

The variable peak costs of purchased energy of 250 €/MWh clearly exceed the variable costs of the gas plant of 60 €/MWh. However the necessary transmission cables for purchasing energy are already needed anyways and do not require additional fix costs in this scenario while the gas plant has total annual fix costs of 183.319 €/MW throughput power and 362.639 €/MW output power. Focussing on one week reducing the needed output capacity by 1MW would save 6.955 €. As showed by the following diagramms this justifies the additional costs of 250 € - 60 € = 190 € per purchased MWh to an amount that reduces the peak load by 73 MW.



Note: For trial e.g. of the result of different storage capacities this `paradiso_3.xlsx` is the input file used for this scenario.

System support by variable prices

Making the prices a function proportional to demand and inversely proportional to intermittent supply is both a good approximation and can demonstrate the system support of such prices. Especially in case of photovoltaics it limits the installed capacity to a reasonable amount and/or encourages investment in storages. This leads to lower peak loads decreasing stress on the grid and a smoother residual demand increasing stability and autarky. Without variable prices storages will run a greedy operation strategy instead of peak shaving and put even more stress on the grid with large power gradients.

Table 18: Scenario Var Cap Sup Im

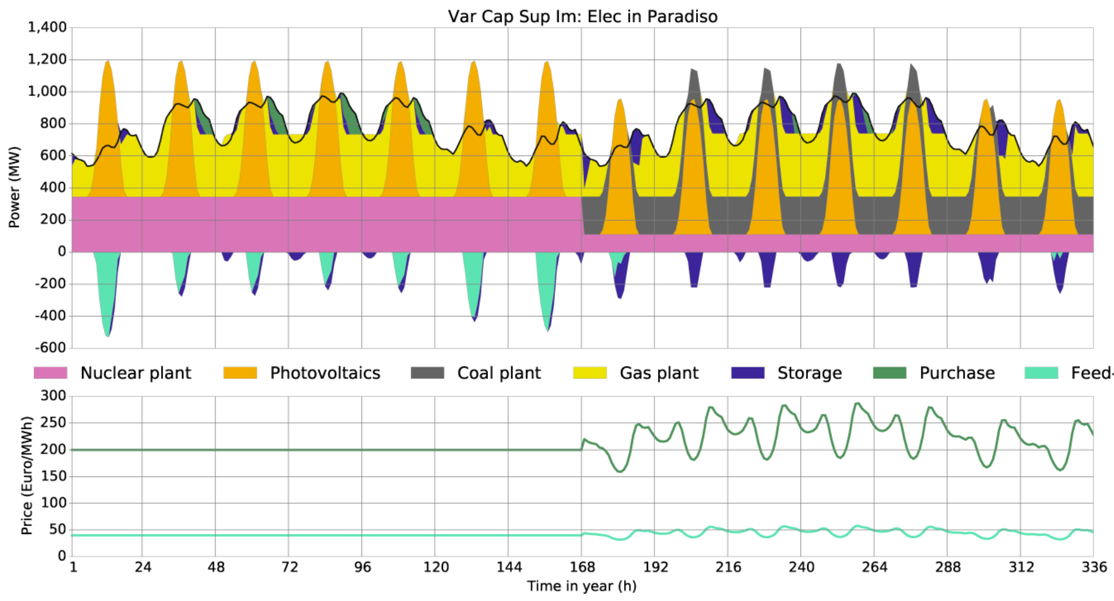
Process	eff	inst-cap	inst-cap-out	fuel-cost	var-cost	total-var-cost
Nuclear plant	0.33	0	0	5	5	10
Coal Plant	0.40	0	0	11	5	16
Gas plant	0.50	0	0	25	5	30
Photovoltaics	1.00	0	0	0	0	0
Storage	1.00	0	0	0	2.5	5
Purchase	1.00	0	0	150-250	0	~200
Feed-in	1.00	0	0	30-50	0	~40

The price function for the scenario was chosen as:

```
Buy price = 100 + 100 * Demand / mean(Demand) * (1.5 - SupIm)

Sell price = Buy Price / 5
```

The result is both more realistic and protective of the grid.



Arbitrage

Arbitrage is the profitable buying and selling of commodities exploiting price differences. For urbs this can be at one timestep or with storages between two different timesteps. It can lead the model to be unbounded, if the buy price at one time step is lower than the sell price or if the price difference between two different timesteps is large enough to finance storage investments. A simple solution to avoid that possibility is to add a large finite upper limit for storage capacities.

1.2.5 Demand Side Management Documentation

This documentation explains the Demand Side Management feature of urbs. With it, one can model time variant Demand Side Management Up/Downshift in a concrete energy system, for example, smart grid of a city.

Introduction

The DSM up/downshifts are closely related to commodities, which are given by default in the urbs with their energy content (MWh). The size of the modelled market has to be considered small relative to the surrounding market. To use this feature, the excel input file needs an additional **Demand Side Management** sheet with the five parameters containing the columns `delay`, `eff`, `recov`, `cap-max-do` and `cap-max-up`, which are used in DSM constraints as technical parameters. For a more detailed description of the implementation have a look at the mathematical definitions in the Mathematical Documentation, section *Demand Side Management Constraints*.

Exemplification

This section contains prototypical scenarios illustrating the system behaviour with time variant DSM up/downshifts. In this part there is an island as an example named `Greenland`, which composed of three sites `Mid`, `North`, and `South`. Between the three sites most of the electricity from `South` has to be transported to supply `Mid`. The electricity of `North` is relatively independent of the other two sites.

When do the electricity DSM downshifts appear in the process?

- it is *necessary* to constraint the whole system with DSM downshifts, if the demand is greater than the total output capacity.
- it is *profitable* to constraint the whole system with DSM downshifts, if the commodity begin to show upward trend till the peak value.

When appears the electricity DSM upshifts in the process?

- it is *possible* **and** *profitable* to constraint the whole system with DSM upshifts, if the demand is lesser than the total output capacity **and** the commodity begin to show downward trend till the valley value.

High Maximal Up/Downshift Capacity

All process, transmission and storage capacities are predetermined and constant.

The following scenario illustrates the energy balance of the `South` of `greenland`. It has a demand of 50-100 GW that is supplied by a 50 GW photovoltaics plant and a 50 GW wind plant. In addition a 50 GW transmission cable exports electricity, which connects the `Mid` of island with the grid of `South`. Both capacities and prices are fix. Because of the meteorological effects on Photovoltaics plants, the timesteps began at the 3000th hour of the year, which was also the beginning of the summer.

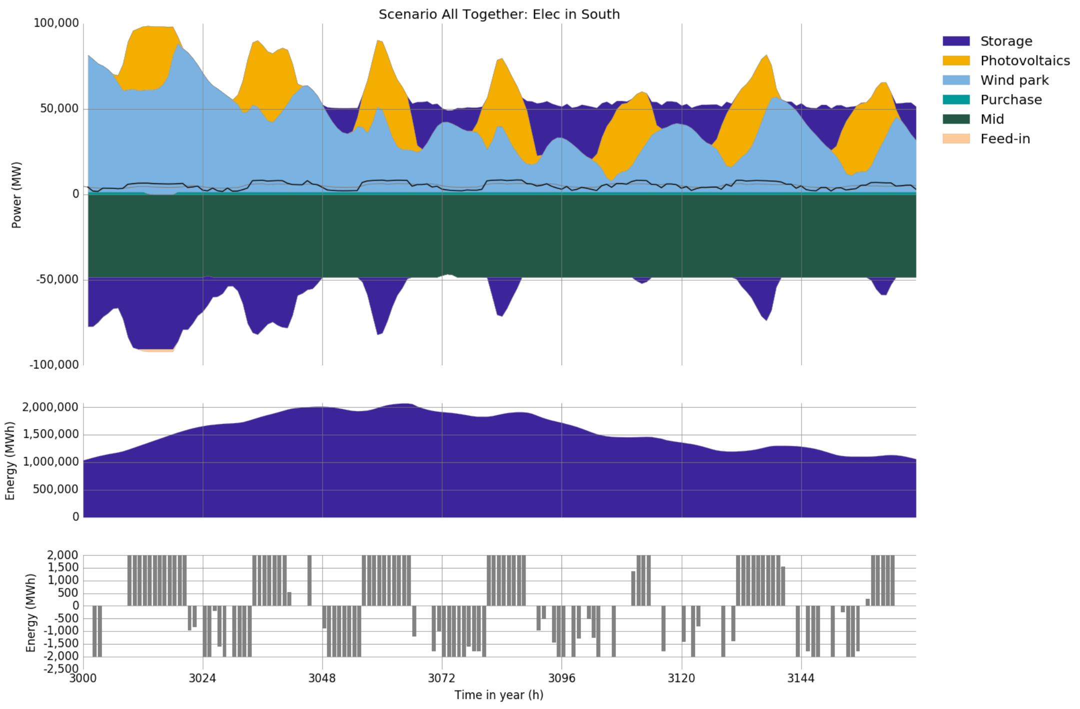
Table 19: Scenario All Together: Elec in South

Process	eff	inst-cap	inst-cap-out	fuel-cost	var-cost	total-var-cost
Photovoltaics	1.00	0	50000	0	0	0
Wind plant	1.00	0	100000	0	0	0
Purchase	1.00	0	1500	**15/45/75**	0	15/45/75
Feed-in	1.00	0	1500	**15/45/75**	0	15/45/75

Table 20: DSM in South

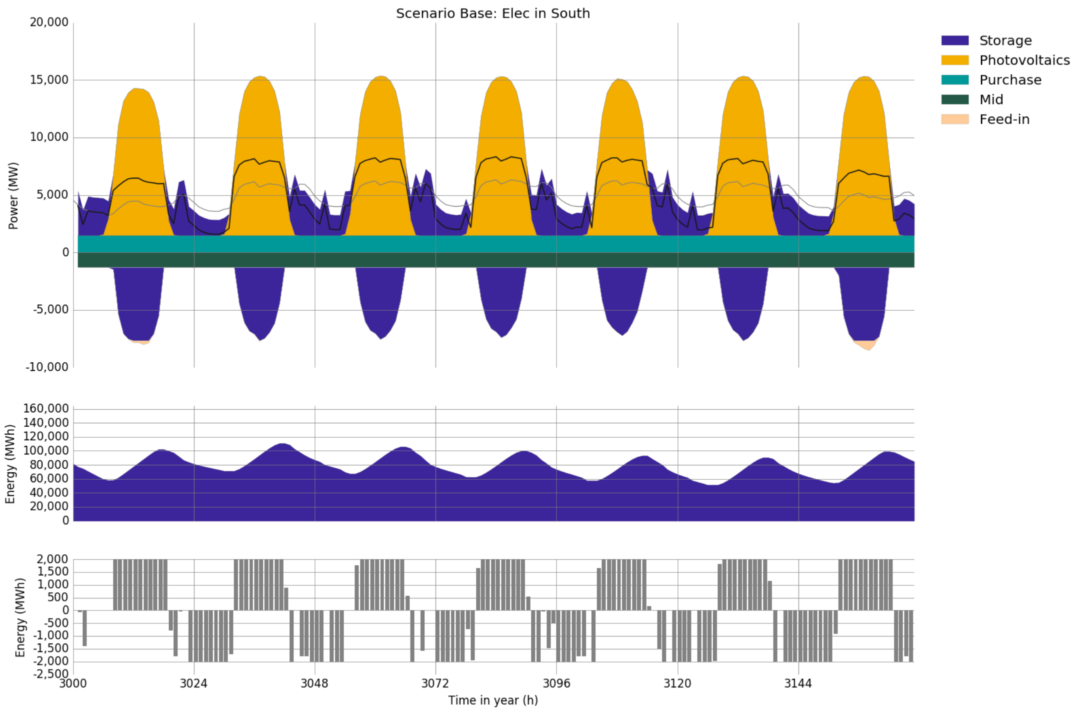
Site	Commodity	delay	eff	recov	cap-max-do	cap-max-up
South	Elec	16	0.90	1	2000	2000

The modelled timesplan lasts 7 days with five parameters from DSM sheet in `greenland-south.xlsx`. In the first ten hours of day 1 the electricity power is at a high level, because the supply is much less than the demand. So the DSM begins with downshifts. But the situation will change into opposite direction over time. After the supply exceeds, the demand the DSM upshifts appears to take place of downshifts. How much electricity can the photovoltaics plants and awind plants generate all depending on the weather conditions. The wind plants works the whole day 24 hours, as long as the wind blows strongly enough. But photovoltaics plants generates electricity only in the daytime, that is why the parameter `delay` is set to 16 hours. It just coincides the time in one day, that is covered by the sunshine. Before the second day the wind blows strongly enough, so that the surplus of wind plant generated electricity is converted into storage. From the 3rd day the wind production decreases, and the electricity of storage has to be taken out to meet the demand. At the midnight of the 5th day electricity capacity come to the lowest point of all, and the output and input keep nearly in balance. Not only the frequency of scenario_base up/downshifts, but also the amount of times of up/downshifts will decrease correspondingly. There is relative more volatility of electricity capacity in the seven days simulation than it without DSM.



If the **commodity stock prices**, **global CO2 limit** and **maximum installable capacity** in `runme.py` are not changed, and just only consider the `scenario_base`, it will be more clearly to show how the

DSM affects the electricity commodities.



Note: For trial e.g. of the result of higher Demand Side Management this `greenland-south.xlsx` is the input file used for this scenario.

Low Maximal Up/Downshift Capacity

All process, transmission and storage capacities are predetermined and constant.

For the second scenario, the `North` of greenland will replaced the `South`. Compared to the `South`, the electricity supply of `North` is relatively simple and independent. It has a demand of 10000-15000 MW, and the supply is dominated by wind plants. Additionally there is about 2500 MW needed to be provided by Purchase.

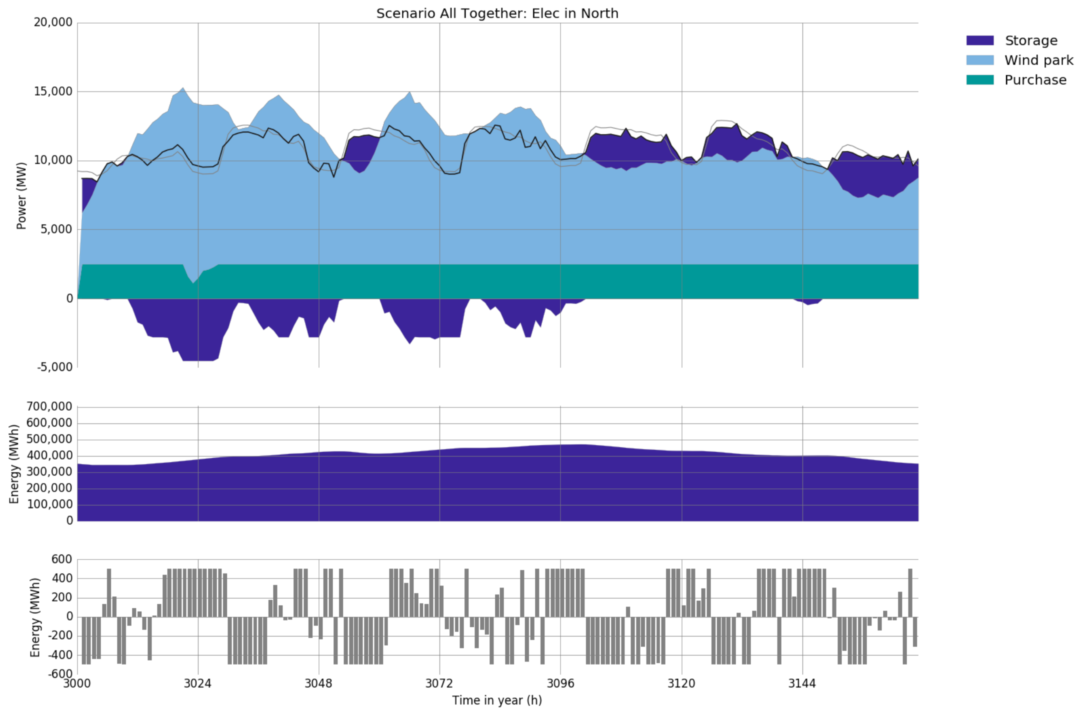
Table 21: Scenario All Together: Elec in North

Process	eff	inst-cap	inst-cap-out	fuel-cost	var-cost	total-var-cost
Photovoltaics	1.00	0	3000	0	0	0
Wind plant	1.00	0	15000	0	0	0
Gas plant	0.60	0	0	27	1.60	28.60
Purchase	1.00	1500	1500	**15/45/75**	0	15/45/75
Feed-in	1.00	2500	2500	**15/45/75**	0	15/45/75

Table 22: DSM in North

Site	Commodity	delay	eff	recov	cap-max-do	cap-max-up
North	Elec	8	1.00	1	500	500

The modelled timesplan lasts also 7 days with five parameters from DSM sheet in greenland-north.xlsx. The electricity supply of North is dominated by wind plants. The wind plants works for 24 hours in one whole day, and the wind power strong or weak has nothing to do with the change of time. So the parameter delay is set to 8 hours. Because the peak value of the output of North is just close to 15 GW, the cap-max-do and cap-max-up are set to 500 MW, which is a quarter of South's. The electricity in the first four days, which is generated by wind plants, keeps at a higher level. That is why the up/downshifts appear frequently, regularly, and alternately during this time. But in the last three days the wind power gets lower, and the electricity of storage has to be taken out to meet the demand. Then during the three days downshifts dominate in most case. With DSM up/downshifts intelligent allocation of electricity resources is required to avoid the shortage of electricity supply during peak hours and the overcapacity in the usual time.



Note: For trial e.g. of the result of lower Demand Side Management this greenland-north.xlsx is the input file used for this scenario.

No Maximal Up/Downshift Capacity

All process, transmission and storage capacities are predetermined and constant.

The last scenario illustrates the energy balance of the Mid of greenland. It has a demand of 50-70 GW that is mostly supplied by a 50 GW transmission, which come from South. In addition, a 13 GW wind plant and 16 GW Photovoltaics plant has made a contribution to the whole electricity system of Mid.

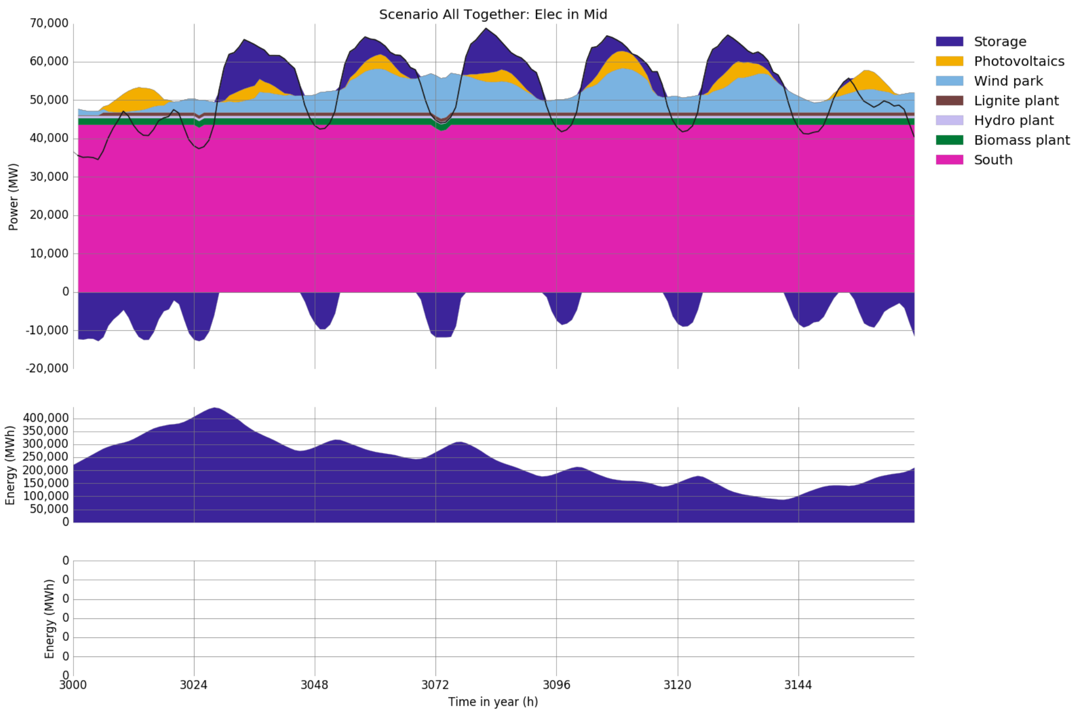
Table 23: Scenario All Together: Elec in Mid

Process	eff	inst-cap	inst-cap-out	fuel-cost	var-cost	total-var-cost
Photovoltaics	1.00	15000	16000	0	0	0.00
Wind plant	1.00	0	13000	0	0	0.00
Gas plant	0.60	0	8000	27	1.60	28.60
Hydro plant	1.00	0	1400	6	1.40	7.40
Lignite plant	0.40	0	60000	0	0.60	0.60
Biomass plant	0.35	0	5000	6	1.40	7.40

Table 24: DSM in Mid

Site	Commodity	delay	eff	recov	cap-max-do	cap-max-up
Mid	Elec	0	1.00	1	0	0

The `Mid` gets so adequate electricity import from the `South`, that commodity of the `Mid` per unit time is far greater than maximal up/downshifts capacity. That means it is meaningless for the setting of DSM faced with so enormous commodity, which is far beyond the controllable range. Supposed that the `Mid` is the city center, the largest energy customer, not the energy producer, and then there is huge infrastructure inside, such as public traffic, hospital, and communication system, which have to be supplied for 24 hours one day. That's why the parameters `delay`, `cap-max-do` and `cap-max-up` are set to 0. It means that there was no more DSM in the electricity system of `Mid` to constraint the commodities.



Note: For trial e.g. of the result of no Demand Side Management this `greenland-mid.xlsx` is the input file used for this scenario.

1.2.6 Decomposition

Overview

How to use the documentation

You should start with this overview which explains the underlying ideas of decomposition in general and the decomposition methods that are used. To fully comprehend the documentation you should be familiar with the urbs model already (see *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 *Tutorial* provides a detailed walkthrough of `runme.py` and explains how to use decomposition for a model. It also explains the Benders loop for each method in detail. After the overview you should continue with the tutorial to understand how to apply the code.

If you want to understand how the decomposition methods work in more detail you should next look at the *Model Class Structure*. This section explains the basic structure of the code and implementation details which are the same for all methods.

The specifics of each decomposition methods model are explained in the sections *Divide Timesteps Model*, *Regional Model* and *SDDP Model*. Refer to these sections to understand where the models differ from the model without decomposition and from each other.

Finally the *Developers 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.

We use three different decomposition methods:

1. Divide Timesteps: Splits the original problem into several time intervals.
2. Regional: Splits the original problem into several regions.
3. SDDP: Splits the original problem into several time intervals, but additionally considers different scenarios for uncertain inputs (e.g. the wind speed).

Benders Decomposition

The idea behind Benders Decomposition is to partition a Linear Program (LP) or Mixed Integer Program (MIP) into several smaller optimization problems.

The LP has the form:

$$\begin{aligned}
min\ & c_0^T \chi_0 + c_1^T \chi_1 \\
s.t.\ & A_0 \chi_0 \geq b_0 \\
& E_0 \chi_0 + A_1 \chi_1 \geq b_1 \\
& \chi_0, \chi_1 \geq 0
\end{aligned}$$

This is done by having a subset of the variables (lets call them χ_0) in a master problem which before the first iteration only contains the constraints depending exclusively on the χ_0 variables. The remaining variable (lets call them χ_1) are given by an unknown future cost function $\eta(\chi_0)$, which is assumed to be constant. The master problem thus looks like this:

$$\begin{aligned} min\ c_0^T\chi_0 + \eta(x) \\ s.t.\ A_0\chi_0 \geq b_0 \\ \chi_0 \geq 0 \end{aligned}$$

This problem is solved and thus gives an optimal solution on χ_0. This solution at the same time gives a lower bound on the optimal objective value (because in later iterations constraints can only be added not removed). The χ_1 variables are optimized in one or several sub problems, which include the constraints on the χ_0 and χ_1 variables. As an example consider two sub problems which split the χ_1 variables into χ_{11} and χ_{12}. This in turn splits the set of constraints A_1 into A_{11} and A_{12} as well as the set E_0 into E_{01} and E_{02}. The sub problems then have the form:

$$\begin{aligned} &min\ \chi_0 + \chi_{11} \\ &s.t.\ A_{11}\chi_{11} \geq b_1 - E_{01}\chi_0 \\ &\chi_1 \geq 0 \end{aligned}$$

and

$$\begin{aligned} &min\ \chi_0 + \chi_{12} \\ &s.t.\ A_{12}\chi_{12} \geq b_1 - E_{02}\chi_0 \\ &\chi_1 \geq 0 \end{aligned}$$

where χ_0 is fixed. Solving the sub problems gives an upper bound on the optimal solution simply by taking the best feasible solution calculated so far in any iteration. Additionally we get a cut we add to the master problem. The cut is a linear function which confines the region of feasible solutions of the master problem. The master problem is then solved again with the cuts as additional constraints. Then the sub problems are solved again using the new optimal values for χ_0. This is repeated until the gap between lower and upper bound gets below a certain threshold.

Divide Timesteps

Splits the original problem into several time intervals at so called support steps.

One sub problem includes the time steps from one support step to the next, including the first support step and excluding the next. The sub instances contain all time dependent variables (all process, transmission and storage variables except capacity). They calculate the optimal value for their variables given restrictions on the capacities by the master problem and in return generate a cut for the master problem.

The master problem contains only the support time steps and optimizes the variables which are time independent (only capacities). It computes an optimal solution based on the cuts given by the sub problems. Using the solution it generates restrictions for the sub problems.

Regional

Splits the original problem into several regions. Here each sub problem consists of one region and contains all the variables and constraints of the original problem in this region. The master problem controls the transmissions between the regions and contains the respective transmission variables.

Additionally a sub problem can be split into regions itself. This can be modelled by passing a separate input file for the sub region. The master problem is oblivious to these sub sub regions and treats the sub region as one. On the other hand this means that the sub problem has to manage its own transmissions including transmissions between its sub sub regions, but also making sure that the transmissions (outgoing, ingoing, and capacities) from the sub sub regions to neighbouring sub regions add up to the same value that the master problem assigned as transmission between the neighbouring sub region and the sub region with the input file. There are some modelling caveats when working with a separate input file. These are explained in *Modelling a region with input file*. The use case for modeling some sub problems with their own file is that for these region additional data is available. If more data is available for all regions it makes sense to have only one input file with a higher resolution, considering the modeling caveats.

SDDP

Splits the original problem into several time intervals, but additionally considers different scenarios for uncertain inputs (e.g. the wind speed). The idea of SDDP is very similar to Divide Timesteps, although the master problem only contains the first time steps for SDDP and not all the support steps. This means that unlike in Divide Timesteps the constraints for the next sub problem are set by the previous problem and not always by the master problem. Likewise the cut is generated for the previous problem.

For each sub problem there are different scenarios (e.g. low wind speed, high wind speed, etc.) called realizations. Each realization is associated with a probability. After the master problem is solved, for each time step a realization is chosen at random and this realization is solved. This gives an optimal solution for one realized path.

This path is used to calculate an upper bound for the objective. As it is unclear if this is indeed a good upper bound due to the uncertainty, we no longer use the difference between upper and lower bound for the convergence criterion, but the difference between the average of the last ten upper bounds plus their standard deviation and the lower bound. This should be a good trade off between using the worst case scenario (e.g. assuming always low wind) which is too pessimistic and using a too low upper bound due to being lucky in choosing a good path.

After the upper bound calculation, a cut is generated for the master problem and for each sub problem except the last. This is done by taking the weighted average of the three cuts generated by the possible realizations in the next sub problem.

Tutorial

This tutorial is a commented walk-through through the script `runme.py`, which is a demonstration user script that can serve as a good basis for ones own script.

In doing this it explains how to apply the decomposition methods and also explains the Benders loop of each decomposition method in detail.

Imports

```
import os
import sys  # save terminal output to file
import time
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
import pyomo.environ
import shutil
import urbs
from urbs import urbsType
from urbs import parallelization as parallel
from pyomo.opt.base import SolverFactory
```

We use the same imports as normal urbs described in *Imports*. PYOMO3 support is not yet included for decomposition. Also we need some additional imports:

- `sys` is a standard python module which we use to redirect the terminal output to a file if desired.
- `time` is used to measure the time for hardware tracking.
- `numpy` and `panda` are python modules which enable fast and convenient handling of data.
- from `urbs` we explicitly import `urbsType` which is used to create different models depending on the decomposition method used and the type of the model (master, sub, sub with input file or normal). `urbs.parallelization` makes it possible to solve several pyomo models in parallel (see *Parallelization*) using the python module `Pyro4`.

Input Settings

We continue with the main function of the script which is the last method in the code.

The script starts with the specification of the input file, which is to be located in the same folder as script `runme.py`:

```
# Choose input file
input_file = 'mimo-example.xlsx'

result_name = os.path.splitext(input_file)[0]  # cut away file extension
result_dir = urbs.prepare_result_directory(result_name)  # name + time␣
↪stamp

# copy input file to result directory
shutil.copyfile(input_file, os.path.join(result_dir, input_file))
# copy runme.py to result directory
shutil.copy(__file__, result_dir)
# copy current version of scenario functions
shutil.copy('urbs/scenarios.py', result_dir)
```

Variable `input_file` defines the input spreadsheet, from which the optimization problem will draw all its set/parameter data. The input file and the script `runme.py` are automatically copied into the result folder.

Next the decomposition method is chosen:

```
# Choose decomposition method (divide-timesteps , regional , sddp or None)
decomposition_method = 'divide-timesteps'
# Check if valid decomposition method is chosen
if decomposition_method not in ['divide-timesteps', 'regional', 'sddp',␣
↪None]:
    raise Exception('Invalid decomposition method. Please choose \'divide-
↪timesteps\' or \'regional\' or \'sddp\' or None')
```

(continues on next page)

(continued from previous page)

Next the desired solver is specified:

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

The solver has to be licensed for the specific user, where the open source solver "glpk" is used as the standard if the solver is not specified.

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

```
# simulation timesteps
(offset, length) = (0, 20)  # time step selection
timesteps = range(offset, offset + length + 1)
```

Variable `timesteps` is the list of time steps 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 time steps to be optimised.

`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]
```

In the next step parameters only specified for certain decomposition methods can be set.

```
# settings for sddp and divide-timesteps
if decomposition_method in ['divide-timesteps', 'sddp']:
    support_steps = [0, 10, 20]

if decomposition_method == 'regional':
    sub_input_files = {'Bavaria': 'bavaria.xlsx'}
```

In Divide Timesteps and SDDP we have to set the support steps which determine at which time steps the problem is split into sub problems. In Divide Timesteps the support steps must include the first and the last time step fpr the method to work correctly. If you don't include them they are added in automatically. In SDDP the last time step is also added automatically to the support steps, but you can choose to omit the first time step. This then means that the input data up to the first support step is considered to be certain and this part is optimized in the master problem. The uncertainty only starts after the first support step. In Regional we can optionally pass a sub input file for any site in a dict of the form `{'site1': 'file1','site2': 'file2'}`.

Scenarios

The `scenarios` list in the end of the runme file allows to select the scenarios to be actually run. How the scenarios are defined and how new ones can be created is explained in *Scenario functions*.

```
scenarios = [
urbs.scenario_base,
urbs.scenario_stock_prices,
```

(continues on next page)

(continued from previous page)

```
urbs.scenario_co2_limit,
urbs.scenario_co2_tax_mid,
urbs.scenario_no_dsm,
urbs.scenario_north_process_caps,
urbs.scenario_all_together]
```

Run scenarios

```
for scenario in scenarios
    result = run_scenario_decomposition(input_file, timesteps, scenario,␣
↪result_dir,
                                            solver=solver,
                                            decomposition_
↪method=decomposition_method,
                                            support_steps=support_steps,
↪# only for divide-timesteps and sddp
                                            sub_input_files={},  # only␣
↪for regional
                                            plot_tuples=plot_tuples,
                                            plot_periods=plot_periods,
                                            report_tuples=report_tuples,
                                            plot_sites_name=plot_sites_
↪name,
                                            report_sites_name=report_sites_
↪name,
                                            plot_and_report=False,
                                            write_lp_files=False,
                                            write_lp_files_every_x_
↪iterations=None,
                                            numeric_focus=False,
                                            save_terminal_output=False,
                                            readable_cuts=False,  # only␣
↪for divide-timesteps
                                            save_hardware_usage=False,
                                            print_omega=False,  # only for␣
↪regional
                                            run_normal=False,
                                            parallel_solving=False,
                                            number_of_workers=None,
                                            save_h5_every_x_
↪iterations=None)
```

Having prepared settings, input data and scenarios, the actual computations happen in the function `run_scenario_decomposition()`. It is executed for each of the scenarios included in the scenario list. The following sections describe the content of function `run_scenario_decomposition()`. 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 results and plots to `result_dir`.

Options of Run Scenario Decomposition

This sub section gives a complete list of the options of `run_scenario_decomposition()` and how to use them.

```
solver=solver,
```

Sets the solver to be used, if None, “glpk” is used.

```
decomposition_method=decomposition_method,
```

Determines the decomposition method. If None, no decomposition is done.

```
support_steps=support_steps,  # only for divide-timesteps and sddp
```

The support steps determine at which points in the time series the original problem is split into sub problems for Divide Timesteps and SDDP.

```
sub_input_files={},  #only for regional
```

In regional it is possible to specify separate input files for sub regions. They are passed in a dict by this option.

```
plot_tuples=plot_tuples,
plot_periods=plot_periods,
report_tuples=report_tuples,
plot_sites_name=plot_sites_name,
report_sites_name=report_sites_name,
plot_and_report=False,
```

All these options except `plot_and_report` are explained in the sections *Plotting* and *Reporting*. If `plot_and_report` is True plotting and reporting is done, if not it is skipped.

```
write_lp_files=False,
write_lp_files_every_x_iterations=None,
```

Debug Feature: If `write_lp_files` is True, the .lp files of the models (contain all information about the model) are saved in a subdirectory of the result directory at the end of the benders loop. If `write_lp_files_every_x_iterations` is set to a natural number, additional .lp files are saved every x iterations. If it is None it is ignored.

```
numeric_focus=False,
```

If `numeric_focus` is True, the solver calculates more carefully. This usually leads to better convergence, but more time spent on solving. The convergence improves especially if the parameters values differ in several orders of magnitude. Therefore it is recommended to use `numeric_focus` whenever convergence is slow.

```
save_terminal_output=False,
```

If True the terminal output is saved to a file inside the result directory.

```
readable_cuts=False,  # only for divide-timesteps
```

Debug Feature: If True, the cuts are represented in a way which makes their mathematical interpretation more clear, but might lead to numerical problems as a multiplication with a number happens which is potentially very close to zero (see *Cut Generation*). Only works for Divide Timesteps.

```
save_hardware_usage=False,
```

Debug/Performance Feature: If True the time and computing resources taken up by the program are saved to a file in the result directory after every iteration of the benders loop.

```
print_omega=False,   # only for regional
```

If True, in the output of each benders iteration of Regional the sum of the omegas is printed. This is in so far interesting as when omega is zero (every 5 iterations) the sub problems are forced to not violate any constraints given by the master problem except the cost constraint. This leads to a faster estimation of an upper bound.

```
run_normal=False,
```

Debug Feature: If True the problem is additionally run without decomposition for comparison.

```
parallel_solving=False,
number_of_workers=None,
```

If `parallel_solving` is True, subproblems are solved in parallel using Pyro where it is possible. In `number_of_workers` the number of Pyro solver servers (MIP servers) can be specified. If it is None the numbers of servers is set to the number of cores by default.

Warning: If you set `parallel_solving` to True make sure that no other programs using Pyro are running, because this could lead to unexpected behaviour or crashes (see *Parallelization*).

```
save_h5_every_x_iterations=None
```

Debug Feature: The solved models are always saved in .h5 files (these contain the models without equations and can be loaded with `urbs.load()`) after convergence of the benders loop. If this option is not None, the models are additionally saved every x iterations.

Complete Walkthrough of Run Scenario Decomposition

```
# This is done as the first thing to get the pyro servers running, so that␣
↪another program can detect another pyro program is running
if parallel_solving:
    # start pyro servers
    servers = parallel.start_pyro_servers(number_of_workers)
```

If `parallel_solving` is True, first the Pyro servers are started up. This is done first, to avoid problems with several Pyro programs running at the same time (see *Parallelization*).

```
# check for valid decomposition method
if decomposition_method not in ['divide-timesteps', 'regional', 'sddp',␣
↪None]:
    raise Exception('Invalid decomposition method. Please choose \'divide-
↪timesteps\' or \'regional\' or \'sddp\' or None')
```

(continues on next page)

(continued from previous page)

Check if decomposition method is valid.

```
# scenario name, read and modify data for scenario
sce = scenario.__name__
data = urbs.read_excel(input_file)
# drop source lines added in Excel
for key in data:
    data[key].drop('Source', axis=0, inplace=True, errors='ignore')
data = scenario(data)
urbs.validate_input(data)
```

Function `read_excel()` 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`.

```
# start saving terminal output to file
if save_terminal_output:
    # save original terminal output to restore later
    write_to_terminal = sys.stdout
    terminal_output_file = open(os.path.join(result_dir, 'terminal-{}.out'.
↪format(sce)), 'w')
    # This class allows to write to the Terminal and to any number of␣
↪files at the same time
    sys.stdout = urbs.TerminalAndFileWriter(sys.stdout, terminal_output_
↪file)
```

The class `TerminalAndFileWriter` in `output.py` redirects the terminal output to both the terminal output and a file. The old value of `sys.stdout` is saved in `write_to_terminal` to be restored later.

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

# setup solver
optim = setup_solver(solver, numeric_focus, logfile=log_filename)
```

Set up the solver.

```
if save_hardware_usage:
    # start_time for hardware tracking
    start_time = time.time()

# create normal
if run_normal or decomposition_method is None:
    prob = urbs.Normal(data, timesteps)
```

(continues on next page)

(continued from previous page)

```
# solve normal
if run_normal or decomposition_method is None:
    result_prob = optim.solve(prob, tee=False)
    print('Original problem objective: ' + str(prob.obj()))

    # save original problem solution (and input data) to HDF5 file
    if run_normal or decomposition_method is None:
        # save models (and input data) to HDF5 file
        h5_dir = os.path.join(result_dir, 'h5_files')
        if not os.path.exists(h5_dir):
            os.makedirs(h5_dir)
        urbs.save(prob, os.path.join(h5_dir, 'original-{}.h5'.format(sce)))

    if write_lp_files:
        lp_dir = os.path.join(result_dir, 'lp_files')
        if not os.path.exists(lp_dir):
            os.makedirs(lp_dir)
        prob.write(os.path.join(lp_dir, 'original' + '-{}.lp'.format(sce)),
                   io_options={'symbolic_solver_labels': True})

if save_hardware_usage:
    track_file = os.path.join(result_dir, scenario.__name__ + '-tracking.
↪txt')
    process = urbs.create_tracking_file(track_file,start_time)
```

If no decomposition method is chosen or `run_normal` is True, solve the original problem and save the solution to a .h5 file.

First the original problem is created by the constructor call to `Normal()`. 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. If `write lp_files` is True, the .lp file is saved. If `save_hardware_usage` is True, the time taken to solve the original problem is measured.

```
# set up models
# set up parameters for divide-timesteps
if decomposition_method == 'divide-timesteps':
    # support time steps
    supportsteps = [i for i in support_steps if i <= max(timesteps)]
    # the support timesteps need to include the max timestep for the␣
↪method to correctly work.
    if not max(timesteps) in supportsteps:
        supportsteps.append(max(timesteps))
    # the support timesteps need to include the min timestep for the␣
↪method to correctly work.
    if not min(timesteps) in supportsteps:
        supportsteps.insert(0,min(timesteps))

    # create models
    master = urbs.DivideTimestepsMaster(data, supportsteps)

    sub = {}
    for inst in range(0, len(supportsteps) - 1):
        sub[supportsteps[inst]+1] = urbs.DivideTimestepsSub(data,␣
↪range(supportsteps[inst], supportsteps[inst + 1] + 1),
        supportsteps)
```

(continues on next page)

(continued from previous page)

```
    # output template
    urbs.create_benders_output_table(print_omega=print_omega)
```

Set up the models and variables specific to the decomposition method Divide Timesteps: First make sure max and min time steps are included in support steps (this is necessary for the method to work correctly). Then create master and sub instances and set up the output table.

```
# set up parameters for regional
elif decomposition_method == 'regional':
    # if 'test_timesteps' is stored in data dict, replace the timesteps␣
↪parameter with that value
    timesteps = data.pop('test_timesteps', timesteps)

    # subproblem data
    sub_data = {}
    for item in sub_input_files:
        sub_data[item] = urbs.read_excel(sub_input_files[item])
        # drop source lines added in Excel
        for key in sub_data[item]:
            sub_data[item][key].drop('Source', axis=0, inplace=True,␣
↪errors='ignore')
        sub_data[item] = scenario(sub_data[item])
        # if 'test_timesteps' is stored in data dict, replace the␣
↪timesteps parameter with that value
        timesteps = sub_data[item].pop('test_timesteps', timesteps)

    # create models
    master = urbs.RegionalMaster(data, timesteps)
    master_sites = urbs.get_entity(master, 'sit')
    master_sites = master_sites.index.values.tolist()

    sub = {}
    for sit in master_sites:
        if sit in sub_input_files:
            sub[sit] = urbs.RegionalSub(sub_data[sit], timesteps, model_
↪type=urbsType.subwfile,
                                        site=sit, msites=master_sites)
        else:
            sub[sit] = urbs.RegionalSub(data, timesteps, model_
↪type=urbsType.sub,
                                        site=sit, msites=master_sites)

    # output template
    urbs.create_benders_output_table(print_omega=print_omega)
```

Similar the models for regional are set up. If separate input files are specified for sub problems they are read into working memory. If only a certain amount of time steps is used for testing, this can be included in the scenario as `test_timesteps`.

```
# set up parameters for sddp
elif decomposition_method == 'sddp':
    # support time steps
    supportsteps = [i for i in support_steps if i <= max(timesteps)]
    # the support timesteps need to include the max timestep for the␣
↪method to correctly work.
```

(continues on next page)

(continued from previous page)

```
    if not max(timesteps) in supportsteps:
        supportsteps.append(max(timesteps))

    # uncertainty factors
    wind_scenarios = {'low': 0, 'mid': 0, 'high': 0}
    realizations = [key for key in wind_scenarios]
    probabilities = {'low': 0.2, 'mid': 0.5, 'high': 0.3}

     # create models
    master = urbs.SddpMaster(data,  range(timesteps[0], supportsteps[0] +␣
↪1), supportsteps, first_timestep=timesteps[0])

    sub = {}
    for inst in range(0, len(supportsteps) - 1):
        for wind_sce in wind_scenarios:
            sub[(supportsteps[inst], wind_sce)] = urbs.SddpSub(data,␣
↪range(supportsteps[inst], supportsteps[inst + 1] + 1),
                supportsteps, uncertainty_factor=wind_scenarios[wind_sce],␣
↪first_timestep=timesteps[0])

    avg = np.inf
    stddev = np.inf
    upper_bounds = []

    #output template
    urbs.create_benders_output_table_sddp()
```

Set up parameters for SDDP. The support steps need to include the last time step for the method to work correctly, but not the first, because the master is allowed to do some of the resource planning. This makes sense, because the time series in the near future can still be considered to be certain, with the uncertainty starting only after some amount of time. `wind_scenarios` are different scenarios on wind speed. `probabilities` give the probability with which a scenario is happening. `avg`, `stddev` are the average and standard deviation of the last ten upper bounds and are used later for the convergence criterion of SDDP. `upper_bounds` is just a list of the calculated upper bounds.

```
if decomposition_method is not None:
    # set up benders loop parameters
    lower_bound = -np.inf
    upper_bound = np.inf
    gap = np.inf
    maxit = 1000
    tol = 1e-6

    # save information for every iteration to plot in the end
    iterations = []
    plot_lower_bounds = []
    plot_upper_bounds = []
    normal = []
```

Set up parameters common to all decomposition methods. These are the initial lower and upper bound and the gap between them, the maximum number of iterations, the tolerance which determines when the Benders loop converges as well as lists of the lower bounds, upper bounds, original objective and iterations for the convergence plot.

```
# call benders loop if a decomposition method is selected
if decomposition_method is not None:
    for i in range(1, maxit):
        # master problem solution
        result_master = optim.solve(master, tee=False)
```

Start of the benders loop (only if decomposition is not None). The loop runs until convergence or until `maxit` is reached. First thing in the loop the current master problem is solved.

```
if decomposition_method == 'divide-timesteps':
    master, sub, lower_bound, upper_bound, gap = benders_loop_divide_
↪timesteps(master, sub, lower_bound, upper_bound, gap, optim, readable_
↪cuts, parallel_solving=parallel_solving)
    # output information about the iteration
    urbs.update_benders_output_table(i, master, sum(master.eta[t]() for t␣
↪in master.tm), sub, lower_bound, upper_bound, gap,   print_omega=print_
↪omega)

elif decomposition_method == 'regional':
    master, sub, lower_bound, upper_bound, gap = benders_loop_
↪regional(master, sub, sub_input_files,
                                                               lower_bound,
↪ upper_bound, gap, optim, i,parallel_solving=parallel_solving)

    # output information about the iteration
    urbs.update_benders_output_table(i, master, sum(master.eta[sit]() for␣
↪sit in master.sit), sub, lower_bound, upper_bound, gap,
                          print_omega=print_omega)

elif decomposition_method == 'sddp':
    master, sub, lower_bound, upper_bound, gap, avg, stddev,  upper_bounds␣
↪= benders_loop_sddp(master, sub, lower_bound, upper_bound, gap,avg,
↪stddev,upper_bounds,supportsteps,
                                                        realizations,␣
↪probabilities, optim, data, first_timestep=timesteps[0], parallel_
↪solving=parallel_solving)

    # output information about the iteration
    urbs.update_benders_output_table_sddp(i, master, lower_bound, upper_
↪bound, avg, stddev, gap, master.obj())
```

The actual loop is different for each decomposition method. The respective functions are explained further down in detail. After the function call, information about the current iteration is printed using the functions `update_benders_output_table(_sddp)()`.

```
if save_hardware_usage:
    # save memory usage
    urbs.update_tracking_file(track_file,i,start_time, process)

# save information for convergence plot
iterations.append(i)
plot_lower_bounds.append(master.obj())
plot_upper_bounds.append(upper_bound)
if run_normal:
    normal.append(prob.obj())
```

If `save_hardware_usage` is True, information about performance of the iteration is saved to the tracking file. The list for the plots are extended by the iteration, the current lower bound, the current upper bound and the original objective (if `run_normal` is True) respectively.

```
if gap < tol * lower_bound:
    #create an excel file which summarizes the results of the benders loop
    if run_normal:
        difference = prob.obj() - master.obj()
        print('\n', 'Difference =', prob.obj() - master.obj())
    else:
        difference = 'Not calculated'
    df = pd.DataFrame([[scenario.__name__, difference, gap, i]],
                      columns=['Scenario', 'Difference', 'gap', 'Iterations
↪'],
                      index=[0])
    urbs.append_df_to_excel(os.path.join(result_dir, 'scenario_comparison.
↪xlsx'), df)
    break
```

The benders loop converges if the gap is smaller than the tolerance times the lower bound. After convergence the difference to the original is calculated if the original was solved. An excel sheet `scenario_comparison.xlsx` is created which contains concise information about the benders convergence for all calculated scenarios.

```
if i % 50 == 0:
    if decomposition_method in ['regional','divide-timesteps']:
        urbs.create_benders_output_table(print_omega=print_omega)
    elif decomposition_method == 'sddp':
        urbs.create_benders_output_table_sddp()
```

For better comprehension of the output table the headline of the output table is repeated every 50 iterations.

```
if save_h5_every_x_iterations is not None and i%save_h5_every_x_iterations␣
↪== 0:
    # save models (and input data) to HDF5 file
    h5_dir=os.path.join(result_dir,'h5_files')
    if not os.path.exists(h5_dir):
        os.makedirs(h5_dir)
    urbs.save(master, os.path.join(h5_dir, 'master' + '-iteration-{}'.
↪format(i) + '-{}.h5'.format(sce)))

    # save subproblems to .h5 files
    for inst in sub:
        urbs.save(sub[inst], os.path.join(h5_dir, 'sub' + str(inst) + '-
↪iteration-{}'.format(i) + '-{}.h5'.format(sce)))

if write_lp_files and write_lp_files_every_x_iterations is not None and i
↪%write_lp_files_every_x_iterations==0:
    # save models to lp files
    lp_dir = os.path.join(result_dir, 'lp_files')
    if not os.path.exists(lp_dir):
        os.makedirs(lp_dir)
    master.write(os.path.join(lp_dir, 'master' + '-iteration-{}'.format(i)␣
↪+ '-{}.lp'.format(sce)),
                 io_options={'symbolic_solver_labels': True})
    for inst in sub:
```

(continues on next page)

(continued from previous page)

```
        sub[inst].write(os.path.join(lp_dir, 'sub' + str(inst) + '-
↪iteration-{}'.format(i) + '-{}.lp'.format(sce)),
                        io_options={'symbolic_solver_labels': True})
```

If the options to save to .h5 or .lp every x iterations are set they are saved in this part of the code. End of benders loop.

```
if parallel_solving:
    # Shut down pyro servers
    parallel.shutdown_pyro_servers(servers)

# close terminal to file stream
if save_terminal_output:
    sys.stdout = write_to_terminal

if plot_and_report:
    # write report to spreadsheet
    urbs.report(
        prob,
        os.path.join(result_dir, '{}.xlsx').format(sce),
        report_tuples=report_tuples, report_sites_name=report_sites_name)

    # result plots
    urbs.result_figures(
        prob,
        os.path.join(result_dir, '{}'.format(sce)),
        timesteps,
        plot_title_prefix=sce.replace('_', ' '),
        plot_tuples=plot_tuples,
        plot_sites_name=plot_sites_name,
        periods=plot_periods,
        figure_size=(24, 9))
```

After the benders loop the Pyro servers are shutdown again (in case of `parallel_solving`) and the terminal output stream is restored (in case of `save_terminal_output`). If `plot_and_report` is True, *Plotting* and *Reporting* is done.

Warning: Plotting and Reporting is so far only supported for the original problem (no decompositon method). If the option `plot_and_report` is True, the decomposition method is not None, and `run_normal` is True, Plotting and Reporting will be done for the normal (not decomposed) problem. If `plot_and_report` is True, the decomposition method is not None, and `run_normal` is False, the program will crash!

```
if decomposition_method is None:
    return prob
else:
    # show plot
    urbs.plot_convergence(iterations, plot_lower_bounds, plot_upper_bounds,
↪ result_dir, sce, run_normal=run_normal, normal=normal)

    # save lp files
    if write_lp_files:
```

(continues on next page)

(continued from previous page)

```
        # save models to lp files
        lp_dir = os.path.join(result_dir, 'lp_files')
        if not os.path.exists(lp_dir):
            os.makedirs(lp_dir)
        master.write(os.path.join(lp_dir, 'master' + '-{}.lp'.format(sce)),
                     io_options={'symbolic_solver_labels': True})
        for inst in sub:
            sub[inst].write(
                os.path.join(lp_dir, 'sub' + str(inst) + '-{}.lp'.
↪format(sce)),
                io_options={'symbolic_solver_labels': True})

    # save models (and input data) to HDF5 file
    h5_dir = os.path.join(result_dir, 'h5_files')
    if not os.path.exists(h5_dir):
        os.makedirs(h5_dir)
    urbs.save(master, os.path.join(h5_dir, 'master' +  '-{}.h5'.
↪format(sce)))

    # save subproblems to .h5 files
    for inst in sub:
        urbs.save(sub[inst],
                  os.path.join(h5_dir, 'sub' + str(inst) +  '-{}.h5'.
↪format(sce)))

    return sub, master
```

If no decomposition method is used the solved instance of the normal model is returned. Otherwise the convergence of the benders loop is shown in a plot and the models solutions are saved in .h5 files and .lp files (only if `write_lp_files` is True). Finally the (solved) sub instances and the master instance are returned.

Walkthrough of Benders Loop Divide Timesteps

```
def benders_loop_divide_timesteps(master, sub, upper_bound, gap, optim, ␣
↪readable_cuts, parallel_solving=False):
    """
    Calculates one iteration of the benders loop for divide timesteps

    Args:
        master: instance of the master problem
        sub: sub problem instances
        upper_bound: current upper bound of benders decomposition
        gap: gap between upper and lower bound
        optim: solver for the problem
        readable_cuts: scale cuts to make them easier to read (may cause␣
↪numerical issues)

    Returns:
        updated values for master, sub, lower_bound, upper_bound, gap
    """

    for inst in sub:
```

(continues on next page)

(continued from previous page)

```
        # restrictions of sub problem
        sub[inst].set_boundaries(master, 'cap_pro', 'pro_inst')
        sub[inst].set_boundaries(master, 'cap_tra', 'tra_inst')
        sub[inst].set_boundaries(master, 'cap_sto_c', 'sto_c_inst')
        sub[inst].set_boundaries(master, 'cap_sto_p', 'sto_p_inst')
        sub[inst].set_boundaries(master, 'e_sto_con', 'e_sto_state')

        sub[inst].eta_res[sub[inst].tm[-1]].expr = master.eta[sub[inst].
↪tm[-1]]()

        for com in master.com_tuples:
            sub[inst].e_co_stock_res[com].expr = master.e_co_
↪stock[sub[inst].tm[-1], com]()
```

First the boundaries of the sub problems are set such that they need to fulfill constraints given by the master problem. Specifically the capacity variables of the sub problem (ending on inst) are set to the capacity given by the master problem, the storage state of the first and last time step of the sub problem are set to the storage content in the corresponding time steps in the master problem, the costs of the sub problem are limited with `eta_res` and the usage of stock commodities is limited by `e_co_stock_res`.

```
if parallel_solving:
    # subproblem solution
    result_sub = parallel.solve_parallel(sub, optim)
else:
    result_sub={}
    for inst in sub:
        # subproblem solution
        result_sub[inst] = optim.solve(sub[inst], tee=False)
```

Next the sub problems are solved. If `parallel_solving` is set, they are passed by the function `solve_parallel` to the running pyro workers (see *Parallelization*). Else they are solved sequentially.

```
# serial cut generation
for inst in sub:
    # cut generation
    master.add_cut(sub[inst], readable_cuts)
```

The cuts are generated and added for each sub problem by a function in the master instance. See *Cut Generation*.

```
lower_bound = master.obj()
```

The optimal solution has to cost at least as much as the current objective of the master problem for the following reasons:

- The master problem objective consists of the costs of a part of the variables (the capacities) which it can optimize and a cost term given by the sub problems which is treated as constant.
- The cost term the master problem can optimize can only get higher in later iterations, because more constraints can be added to the master problem, but no constraints can be removed.
- The costs given by the sub problems can only get higher, because the bounds the sub problems receive from the master problem can only get tighter as the master problem acquires more cuts.

```
try:
    # Check feasibility of subproblems with respect to constraints for␣
↪which additional cost cannot be computed
    for inst in sub:
        for ct in sub[inst].com_tuples:
            if sub[inst].commodity.loc[ct, 'max'] < np.inf:
                if sum(sub[inst].e_co_stock[(tm,) + ct]() for tm in␣
↪sub[inst].tm) - sub[inst].e_co_stock_res[ct]() > 0.001:
                    raise ValueError("Subproblem violates stock commodity␣
↪constraints!")

        for sit, sto, com in sub[inst].sto_tuples:
            for t in sub[inst].tm:
                if t == sub[inst].ts[1]:
                    if (sub[inst].e_sto_con[t, sit, sto, com]() -
                            sub[inst].e_sto_state[t, sit, sto, com]() > 0.
↪001):
                        raise ValueError("Subproblem violates storage␣
↪content constraints!")
                if t == sub[inst].ts[2]:
                    if (sub[inst].e_sto_con[t, sit, sto, com]() -
                            sub[inst].e_sto_state[t, sit, sto, com]() < -0.
↪001):
                        raise ValueError("Subproblem violates storage␣
↪content constraints!")

        if sub[inst].dt * sub[inst].weight * sum(- urbs.modelhelper.
↪commodity_balance(sub[inst], tm, sit, 'CO2')()
                                                 for tm in sub[inst].tm
                                                 for sit in sub[inst].sit) \
                - sum(sub[inst].e_co_stock_res[sit, 'CO2', 'Env']() for␣
↪sit in sub[inst].sit) > 0.001:
            raise ValueError("Subproblem violates CO2 constraints!")
```

Try if any of the sub problems violates any of the following constraints:

- Stock commodity constraints: Violated if the sub problems uses more of a commodity than it is given by the master problem.
- Storage content constraints: Violated if any of the sub problems storages is greater than the storage assigned to it by the master problem in the first time step or lower than the storage it needs to have left in the last time step.
- CO2 constraints: Violated if the maximum allowed threshold for CO2 is passed.

If one of the constraints is violated, the sub problem is infeasible. In this case we cannot compute an upper bound in this iteration.

```
# determining the costs of units' production between iterations
cost_pro = urbs.get_production_cost(master, sub, 'cap_pro', 'pro')
cost_sto_c = urbs.get_production_cost(master, sub, 'cap_sto_c', 'sto_c')
cost_sto_p = urbs.get_production_cost(master, sub, 'cap_sto_p', 'sto_p')

cost_tra = 0.0

for sin, sout, type, com in master.tra_tuples:
    max_tra = max(max(sub[inst].e_tra_in[(tm, sin, sout, type, com)]()
```

(continues on next page)

(continued from previous page)

```
                        for inst in sub
                        for tm in sub[inst].tm),
                    max(sub[inst].e_tra_in[(tm, sout, sin, type, com)]()
                        for inst in sub
                        for tm in sub[inst].tm))
    tra = (sin, sout, type, com)
    if max_tra > master.cap_tra[tra]():
        cost_tra += ((max_tra - master.cap_tra[tra]()) *
                     master.transmission.loc[tra]['inv-cost'] *
                     master.transmission.loc[tra]['annuity-factor'])

costs = cost_pro + cost_tra + cost_sto_c + cost_sto_p
```

Check if for any process, storage, or transmission variable in the sub problems the capacity is higher than the capacity installed in the master problem. If this happens the master problem needs to install the maximum capacity needed for that variable in any sub problem (This is done by the function `get_production_cost()`. See *Benders Functions*). The cost of this installation is accumulated in costs.

```
# convergence check
gap, lower_bound, upper_bound = urbs.convergence_check(master, sub, upper_
↪bound, costs, 'divide-timesteps')
```

Calculate the new upper bound and gap (see *Benders Functions*).

```
except ValueError as err:
    print("Upper bound not updated because subproblem constraints were␣
↪violated! (" + str(err) + ")")
return master, sub, lower_bound, upper_bound, gap
```

Except the ValueError if no upper bound was calculated and return the updated models and values.

Walkthrough of Benders Loop Regional

```
def benders_loop_regional(master, sub, sub_input_files, lower_bound, upper_
↪bound, gap, optim, i, parallel_solving=False):
    """
    Calculates one iteration of the benders loop for regional

    Args:
        master: instance of the master problem
        sub: sub problem instances
        sub_input_files: list of filenames to Excel spread sheets for sub␣
↪regions, can be set for regional method
        lower_bound: current lower bound of benders decomposition
        upper_bound: current upper bound of benders decomposition
        gap: gap between upper and lower bound
        optim: solver for the problem
        i: number of the current iteration
        parallel_solving: If true sub instances are solved in parallel␣
↪with pyro

    Returns:
```

(continues on next page)

(continued from previous page)

```
        updated values for master, sub, lower_bound, upper_bound, gap,␣
↪track_file
    """
    if i % 5 == 0:
        for inst in sub:
            getattr(sub[inst], 'omega').set_value(0)
    else:
        for inst in sub:
            getattr(sub[inst], 'omega').set_value(1)
```

Every five iterations omega is set to zero. As a consequence the sub problems are forced to not violate any constraints given by the master problem except the cost constraint. This leads to a faster estimation of an upper bound, because the sub problem becomes feasible as no constraints can be violated.

```
# subproblem restrictions
for inst in sub:
    # subproblem with input file
    if inst in sub_input_files:
        # e_co_stock
        for tm in master.tm:
            sub[inst].e_co_stock_res[tm] = master.e_co_stock[tm, sub[inst].
↪sub_site[1], 'CO2', 'Env']()
        # cap_tra
        for tra in master.tra_tuples:
            if tra[0] == sub[inst].sub_site[1]:
                sub[inst].hvac[tra[1]] = master.cap_tra[tra]()
            else:
                continue
        # e_tra
        for tm in master.tm:
            for tra in master.tra_tuples:
                if tra[0] == sub[inst].sub_site[1]:
                    sub[inst].e_export_res[tm, tra[1]] = master.e_tra_
↪out[tm, tra]()
                elif tra[1] == sub[inst].sub_site[1]:
                    sub[inst].e_import_res[tm, tra[0]] = master.e_tra_
↪in[tm, tra]()
                else:
                    continue
        # eta
        sub[inst].eta_res[sub[inst].sub_site[1]] = master.eta[sub[inst].
↪sub_site[1]]()
    else:
        sub[inst].set_boundaries(master, 'e_co_stock', 'e_co_stock_res')
        sub[inst].set_boundaries(master, 'e_tra_out', 'e_tra_out_res')
        sub[inst].set_boundaries(master, 'e_tra_in', 'e_tra_in_res')
        sub[inst].set_boundaries(master, 'eta', 'eta_res')
```

Set the boundaries of the sub problems to fulfill constraints given by the master problem. For both sub problems with and without input files we set the restrictions on the cost `eta_res` to the cost given by the master problem and the stock commodity restriction `e_co_stock_res` (only relevant for CO2). In case of sub with input file we need to set everything using `sub_site[1]` which just represents the name of the site in the master problem.

For a sub problem without input file only we set the restriction on the in- and outgoing transmissions `e_tra_in_res` and `e_tra_out_res`.

For a sub problem with input file we need to explicitly set the boundaries on transmission capacity (hvac), import and export. For hvac and export we take all transmissions tuples that originate in the sub problem (tra[0] == sub[inst].sub_site[1]) and set hvac to the capacity and e_export_res to the outgoing transmission. For e_import_res we do the same as for export, but checking for incoming transmissions (tra[1] == sub[inst].sub_site[1]).

```
# sub problem solution
if parallel_solving:
    result_sub = parallel.solve_parallel(sub, optim)
else:
    result_sub={}
    for inst in sub:
        result_sub[inst] = optim.solve(sub[inst], tee=False)
```

Next the sub problems are solved. If parallel_solving is set, they are passed by the function solve_parallel() to the running Pyro workers (see *Parallelization*). Else they are solved sequentially.

```
# serial cut generation
for inst in sub:
    # cut generation
    if inst in sub_input_files:
        master.add_cut(sub[inst],sub_in_input_files=True)
    else:
        master.add_cut(sub[inst], sub_in_input_files=False)
```

The cuts are generated and added to the master for each sub problem by a function in the master instance (see *Cut Generation*).

```
# convergence check
if i % 5 == 0:
    gap, lower_bound, upper_bound = urbs.convergence_check(master, sub,␣
↪upper_bound, 0, 'regional')

return master, sub, lower_bound, upper_bound, gap
```

Update lower and upper bound and return (see *Benders Functions*).

Walkthrough of Benders Loop SDDP

```
def benders_loop_sddp(master, sub, lower_bound, upper_bound, gap, avg,␣
↪stddev,upper_bounds, supportsteps, realizations, probabilities,
                      optim, data, first_timestep=0, parallel_solving=False):
    """
    Calculates one iteration of the benders loop for regional

    Args:
        master: instance of the master problem
        sub: sub problem instances
        lower_bound: current lower bound of the benders decomposition
        upper_bound: current upper bound of the benders decomposition
        gap: gap between lower and upper bound
        avg: average of the last 10 upper bounds
        stddev: standard deviation within the last 10 upper bounds
```

(continues on next page)

(continued from previous page)

```
        upper_bounds: list of upper bounds
        supportsteps: a list of timesteps for the master problem, can be␣
↪set for divide-timesteps method
        realizations: dict of possible realizations of sub problems (e.g.
↪'high', 'mid', 'low')
        probabilities: probabilities of the realizations
        optim: solver for the problem
        data: The data given by the input file.
        parallel_solving: If true, the possible realizations in the␣
↪backward iteration are solved in parallel
        first_timestep: The timestep at which the non decomposed problem␣
↪starts. This is needed to calculate the weight parameter correctly. The␣
↪default is set to 0.

    Returns:
        updated values for master, sub, lower_bound, upper_bound, gap
    """


    # dict for realized instances of sub
    realize={}
    # Forward recursion
    for inst in range(0, len(supportsteps) - 1):
        realize[inst] = np.random.choice(realizations, p=[value for value␣
↪in probabilities.values()])

        # save current problem
        cur_prob = sub[(supportsteps[inst], realize[inst])]

        # if previous problem is the master problem
        if inst == 0:
            # set previous problem
            prev_prob = master
        else:
            prev_prob = sub[(supportsteps[inst - 1], realize[inst - 1])]
```

In the forward recursion we pick a realization of each sub problem at random and set the previous problem to the realized instance of the previous sub problem or to the master problem in case of the first subproblem.

```
# exchange variables between time steps
 cur_prob.set_boundaries(prev_prob, 'cap_pro', 'pro_inst')
 cur_prob.set_boundaries(prev_prob, 'cap_tra',  'tra_inst')
 cur_prob.set_boundaries(prev_prob, 'cap_sto_c',  'sto_c_inst')
 cur_prob.set_boundaries(prev_prob, 'cap_sto_p',  'sto_p_inst')
 cur_prob.set_boundaries(prev_prob, 'e_sto_con',  'e_sto_con_res')
 cur_prob.set_boundaries(prev_prob, 'e_co_stock_state',  'e_co_stock_state_
↪res')

 if inst > 0:
     cur_prob.eta_res.expr = prev_prob.eta()

 # solve problem
 optim.solve(cur_prob, tee=False)
```

Set the constraints on the capacities, the storage content and the stock reserves to the values passed by the previous problem. Also set the constraint on the costs (eta_res) to the value given by the previous sub problem. In case of the first sub problem we do not need to set this constraint, because the master problem contains only the zero-th time step and thus does not contribute any cost restriction.

Then the sub problem is solved. End of the forward recursion.

```
# update upper bound
try:
    # Check feasibility of subproblems with respect to constraints for␣
↪which additional cost cannot be computed
    max_value = {}
    violation = {}
    violation_factor = 0.0001

    for sub_inst in [sub[(supportsteps[inst], realize[inst])] for inst in␣
↪range(0, len(supportsteps) - 1)]:
        for (sit, com, com_type) in sub_inst.com_max_tuples:
            try:
                max_value[(sit, com, com_type)] += sub_inst.e_co_stock_
↪state[
                                                 sub_inst.t[-1], sit,
↪ com, com_type]() \
                                             - sub_inst.e_co_stock_
↪state[
                                                 sub_inst.t[1], sit,␣
↪com, com_type]()
            except KeyError:
                max_value[(sit, com, com_type)] = sub_inst.e_co_stock_
↪state[
                                                 sub_inst.t[-1], sit,
↪ com, com_type]() \
                                             - sub_inst.e_co_stock_
↪state[
                                                 sub_inst.t[1], sit,␣
↪com, com_type]()
```

Calculate the maximum used value of all commodities in all sites. The value is simply calculated by taking the sum of what each sub problem uses of the commodity in the site. How much the sub problem needs is calculated by taking the commodity stock at the last time step minus the commodity stock in the first time step.

```
weight = master.weight()
max_output_ratio_elec_co2 = (master.r_out.xs('Elec', level=1) / master.r_
↪out.xs('CO2', level=1).loc[master.r_out.xs('CO2', level=1) != 0]).
↪replace(np.inf,np.nan).max()
costs_co2_violation = 0
violation_bound = violation_factor * data['commodity'].loc[sit, com, com_
↪type]['max']
for (sit, com, com_type) in max_value.keys():
    violation[(sit, com, com_type)] = max_value[(sit, com, com_type)] *␣
↪weight - \
                                      data['commodity'].loc[sit, com, com_
↪type]['max']

    if violation[(sit, com, com_type)] > violation_bound:
        raise ValueError(f"Path violates maximum commodity constraint! (
↪{violation[(sit, com, com_type)]})")
```

(continues on next page)

(continued from previous page)

```
    elif violation[(sit, com, com_type)] > violation_bound*0.01:
        # determining violation costs for commodity violation in case of␣
↪co2
        if com == 'CO2':
            co2_costs = max_output_ratio_elec_co2 * violation[(sit, com,␣
↪com_type)] * \
                        master.commodity.loc[sit, 'Slack', 'Stock']['price
↪'] * weight
            costs_co2_violation += co2_costs
        else:
            raise ValueError(f"Path violates maximum commodity constraint!"
                             f"({violation[(sit, com, com_type)]})")

violation_bound = violation_factor * data['global_prop'].loc['CO2 limit',
↪'value']
if sum(max_value[(sit, com, com_type)] for (sit, com, com_type) in max_
↪value.keys() if
       com_type == 'Env') * weight - data['global_prop'].loc['CO2 limit',
↪'value'] > violation_bound:
    raise ValueError(f"Path violates global environmental rule!"
                     f" ({sum(max_value[(sit, com, 'Env')] for (sit, com,␣
↪com_type) in max_value.keys()) * weight}")
```

Try if any of the sub problems violates any of the following constraints.

- Constraint 1: Check if the maximum used value of any commodity is more than 0.01 times the violation bound greater than the maximum allowed amount of that commodity. For all commodities except CO2 this triggers an exception. In case of CO2 the exception is only triggered if the violation is more than the violation bound. If it is between the violation bound and 0.01 times the violation bound we compute a violation cost which is taken to be as high as the cost of producing Slack “energy”. The idea of this is to get a faster estimate of an upper bound, because the CO2 constraint is often violated. To estimate a cost for the CO2 violation we replace a power plant that produces the most electricity per CO2 (`max_output_ratio_elec_co2`) and replace it with an expensive Slack power plant that doesn’t produce CO2.
- Constraint 2: Check if the sum of environmental commodities exceeds the allowed CO2 limit by more than the violation bound.

```
# determining violation costs for storage content
costs_sto_violation = 0

for sub_inst in [sub[(supportsteps[inst], realize[inst])] for inst in␣
↪range(0, len(supportsteps) - 1)]:
    for sit, sto, com in sub_inst.sto_tuples:
        for t in sub_inst.ts:
            if t == sub_inst.ts[1]:
                if (sub_inst.e_sto_con[t, sit, sto, com]() -
                        sub_inst.e_sto_con_res[t, sit, sto, com]() > 1):
                    raise ValueError(f"Subproblem violates storage content␣
↪constraints!"
                                     f"{sub_inst.e_sto_con[t, sit, sto,␣
↪com]() - sub_inst.e_sto_con_res[t, sit, sto, com]()}")
                elif (sub_inst.e_sto_con[t, sit, sto, com]() -
                      sub_inst.e_sto_con_res[t, sit, sto, com]() > 0.01):
                    costs_sto_violation += (sub_inst.e_sto_con[t, sit, sto,
↪ com]() - sub_inst.e_sto_con_res[t, sit, sto, com]()) \
```

(continues on next page)

(continued from previous page)

```
                                        * sub_inst.commodity.loc[sit,
↪'Slack', 'Stock']['price'] * weight

sub_inst = sub[(supportsteps[-2], realize[len(supportsteps) - 2])]
t_end = sub_inst.t[-1]
t_start = master.t[1]
start_end_difference = master.e_sto_con[t_start, sit, sto, com]() - sub_
↪inst.e_sto_con[t_end, sit, sto, com]()
violation_bound = violation_factor * master.e_sto_con[t_start, sit, sto,␣
↪com]()
for sit, sto, com in sub_inst.sto_tuples:
    if start_end_difference > violation_bound:
        raise ValueError(
            f"Subproblem violates storage content start end constraints!"
            f"{start_end_difference}")
    elif (start_end_difference > violation_bound*0.1):
        costs_sto_violation += start_end_difference \
                               * sub_inst.commodity.loc[sit, 'Slack',
↪'Stock']['price'] * weight
```

Next we calculate the costs for storage violations: First we check for every sub problem whether it fulfills its storage usage constraint. In case it exceeds its limitation by more than one we throw an error. In case it exceeds it by more than 0.01 we assume the cost of producing the energy deficit as Slack energy. Next we have to check whether the storage content in the first time step is bigger than the storage content in the last time step ("storage content start end constraints"). As we require our problem to leave as much energy in the storage as it started with this throws an error.

```
# determining the costs of units' production between iterations
worst_case_realization = 'low'

additional_costs = {}
cost_types = ['pro', 'sto_c', 'sto_p']

for ctype in cost_types:
    additional_costs[ctype] = max(urbs.get_production_cost(master,
                                        {(supportsteps[inst], worst_case_
↪realization): sub[
                                            (supportsteps[inst], worst_
↪case_realization)] for inst in
                                         range(0, len(supportsteps) - 1)},
                                        f'cap_{ctype}', ctype),
               urbs.get_production_cost(master,
                                        {(supportsteps[inst],␣
↪realize[inst]): sub[
                                            (supportsteps[inst],␣
↪realize[inst])] for inst in
                                         range(0, len(supportsteps) - 1)},
                                        f'cap_{ctype}', ctype)
               )

cost_tra = 0.0

for sin, sout, type, com in master.tra_tuples:
    max_tra = max(max(sub_inst.e_tra_in[(tm, sin, sout, type, com)]()
                      for sub_inst in
```

(continues on next page)

(continued from previous page)

```
                        [sub[(supportsteps[inst], realize[inst])] for inst␣
↪in range(0, len(supportsteps) - 1)]
                        for tm in sub_inst.tm),
                    max(sub_inst.e_tra_in[(tm, sout, sin, type, com)]()
                        for sub_inst in
                        [sub[(supportsteps[inst], realize[inst])] for inst␣
↪in range(0, len(supportsteps) - 1)]
                        for tm in sub_inst.tm))
    tra = (sin, sout, type, com)
    if max_tra > master.cap_tra[tra]():
        cost_tra += ((max_tra - master.cap_tra[tra]()) *
                     master.transmission.loc[tra]['inv-cost'] *
                     master.transmission.loc[tra]['annuity-factor'])

# sum up all additional costs
costs = cost_tra + costs_sto_violation + costs_co2_violation +␣
↪sum(additional_costs.values())
```

We also need to check whether the sub problems use more of any capacity than the master problem has installed. If so we need to add the cost of installing the needed capacities. This cost can be calculated for the process and storage variables using the function `get_production_cost()` (see *Benders Functions*). The transmission cost is calculated slightly different. We then add up all costs.

```
upper_bound = (master.obj() - master.eta() + costs
               + sum(sub[(supportsteps[inst], realize[inst])].costs[cost_
↪type]()
               for cost_type in ["Variable", "Fuel", "Environmental"]
               for inst in range(0, len(supportsteps) - 1)))

upper_bounds.append(upper_bound)
```

We update the current upper bound by summing up the master cost (`master.obj()`) minus the old costs of the subproblems (`master.eta()`) plus the additional investment costs accumulated in costs plus the new costs of the sub problems. The new upper bound is appended to the list of upper bounds.

```
if len(upper_bounds) > 10:
    bounds = upper_bounds[-10:]
    avg = np.mean(bounds)
    stddev = np.std(bounds)
    gap = avg + 1 * stddev - lower_bound
```

If more than ten upper bounds have been calculated, we take the average and the standard deviation of the last ten and use this to calculate the new gap by taking the average plus the standard deviation minus the lower bound (see overview of *SDDP*).

```
except ValueError as err:
    print("Upper bound not updated because subproblem constraints were␣
↪violated! (" + str(err) + ")")
```

If no upper bound was calculated print which constraint was violated.

```
# Backward recursion
for inst in range(len(supportsteps) - 2, -1, -1):
    # if previous problem is the master problem
```

(continues on next page)

(continued from previous page)

```
    if inst == 0:
        # set previous problem
        prev_prob = master
    else:
        prev_prob = sub[(supportsteps[inst - 1], realize[inst - 1])]

    cur_probs = {}
    for cur_real in realizations:
        cur_prob = sub[(supportsteps[inst], cur_real)]

        # exchange variables between time steps
        cur_prob.set_boundaries(prev_prob, 'cap_pro', 'pro_inst')
        cur_prob.set_boundaries(prev_prob, 'cap_tra', 'tra_inst')
        cur_prob.set_boundaries(prev_prob, 'cap_sto_c', 'sto_c_inst')
        cur_prob.set_boundaries(prev_prob, 'cap_sto_p', 'sto_p_inst')
        cur_prob.set_boundaries(prev_prob, 'e_sto_con', 'e_sto_con_res')
        cur_prob.set_boundaries(prev_prob, 'e_co_stock_state', 'e_co_stock_
↪state_res')

        cur_prob.eta_res.expr = prev_prob.eta()

        cur_probs[(supportsteps[inst],cur_real)] = cur_prob
```

In the backward recursion we calculate a cut for the master problem and for all realizations of all sub problem except the ones in the last time step (outer for-loop). To do this we take the weighted (by the scenario probability) average of the cuts generated by the realizations of the next sub problem. As we so far only solved one realization, we now have to solve all realizations of all sub problems. To do this we first set the boundaries like in the forward iteration, but for all realizations. We append all realizations of one sub problem to the dict `cur_probs`.

```
# solve realizations
if parallel_solving:
    # subproblem solution
    parallel.solve_parallel(cur_probs, optim)
else:
    for cur_prob in cur_probs:
        # subproblem solution
        optim.solve(cur_probs[cur_prob], tee=False)
```

Solve the realizations in `cur_probs`. The problems can be solved in parallel (see *Parallelization*).

```
# cut generation
cut_generating_problems = {}
for cur_real in realizations:
    cut_generating_problems[cur_real] = sub[supportsteps[inst], cur_real]
if inst == 0:  # prev_prob is the master problem
    prev_prob_realize = master
    prev_prob = master
    prev_prob.add_cut(realizations, cut_generating_problems, prev_prob_
↪realize, probabilities)

else:
    prev_prob_realize = sub[supportsteps[inst - 1], realize[inst - 1]]
    for prev_real in realizations:
        prev_prob = sub[supportsteps[inst - 1], prev_real]
```

(continues on next page)

(continued from previous page)

```
        prev_prob.add_cut(realizations, cut_generating_problems, prev_prob_
↪realize, probabilities)
```

To every possible realization of the current instance we add a weighted cut using the function `add_cut()`. The weighted cut consists of one cut for each realization in the next time step (`cut_generating_problems`) weighted by their probability. See SDDP *Cut Generation*. End of the backward iteration.

```
lower_bound = master.obj()
```

Update the lower bound. The optimal solution has to cost at least as much as the current objective of the master problem for the following reasons:

- The master problem objective consists of the costs of a part of the variables (the capacities) which it can optimize and a cost term given by the sub problems which is treated as constant.
- The cost term the master problem can optimize can only get higher in later iterations, because more constraints can be added to the master problem, but no constraints can be removed.
- The costs given by the sub problems can only get higher, because the bounds the sub problems receive from the master problem can only get tighter as the master problem acquires more cuts.

```
return master, sub, lower_bound, upper_bound, gap, avg, stddev,  upper_
↪bounds
```

Return the updated problem instances and bounds.

Output

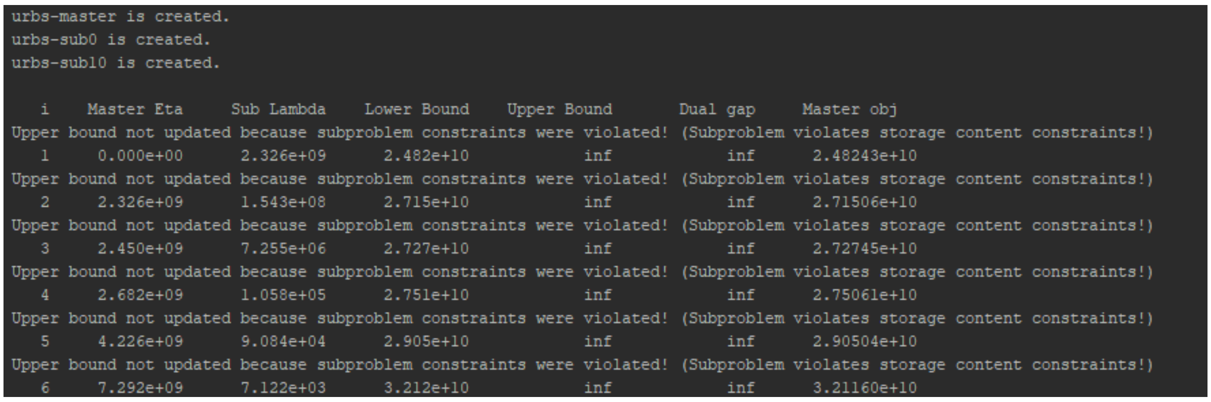
All functions related to output are in the file `output.py` in the urbs directory. All outputs are saved to the result directory which is created by the function `prepare_result_directory()`.

Terminal Output

The terminal output consists of information about which models are created, the normal's objective (if the normal is run) and information about each iteration of the bender's loop (if decomposition is run). If both are run, it also contains the difference between the normal's and the master's objective. The functions to output information about the benders loop are:

- `create_benders_output_table()` and `create_benders_output_table_sddp()` to write the headline
- `update_benders_output_table()` and `update_benders_output_table_sddp()` to output the information about the iteration.

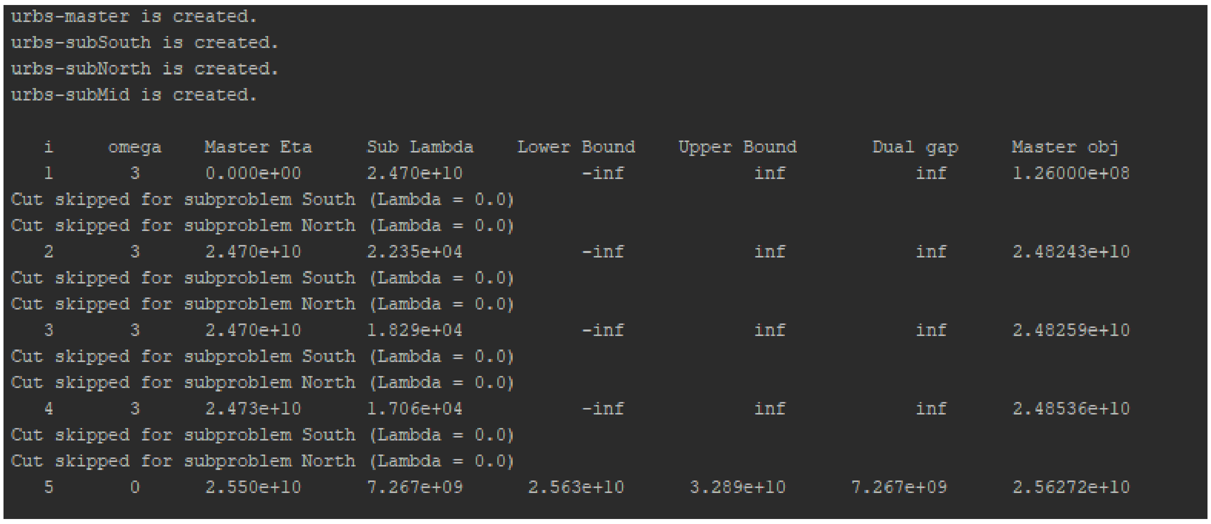
The terminal output can be saved to a file `terminal-scenario_name.out` by setting the option `save_terminal_output` to True.



```
urbs-master is created.
urbs-sub0 is created.
urbs-sub10 is created.

   i    Master Eta     Sub Lambda    Lower Bound    Upper Bound       Dual gap     Master obj
Upper bound not updated because subproblem constraints were violated! (Subproblem violates storage content constraints!)
   1     0.000e+00      2.326e+09      2.482e+10           inf           inf      2.48243e+10
Upper bound not updated because subproblem constraints were violated! (Subproblem violates storage content constraints!)
   2     2.326e+09      1.543e+08      2.715e+10           inf           inf      2.71506e+10
Upper bound not updated because subproblem constraints were violated! (Subproblem violates storage content constraints!)
   3     2.450e+09      7.255e+06      2.727e+10           inf           inf      2.72745e+10
Upper bound not updated because subproblem constraints were violated! (Subproblem violates storage content constraints!)
   4     2.682e+09      1.058e+05      2.751e+10           inf           inf      2.75061e+10
Upper bound not updated because subproblem constraints were violated! (Subproblem violates storage content constraints!)
   5     4.226e+09      9.084e+04      2.905e+10           inf           inf      2.90504e+10
Upper bound not updated because subproblem constraints were violated! (Subproblem violates storage content constraints!)
   6     7.292e+09      7.122e+03      3.212e+10           inf           inf      3.21160e+10
```

Here you can see the terminal output of Divide Timesteps. Information is printed about the masters future costs (Master Eta), the sum of the sub problems `Lambda` (Sub Lambda), the lower and upper bound, the gap between them, and the master objective which is equal to the lower bound. Additionally the output informs you if the upper bound is not updated and what constraint was violated.

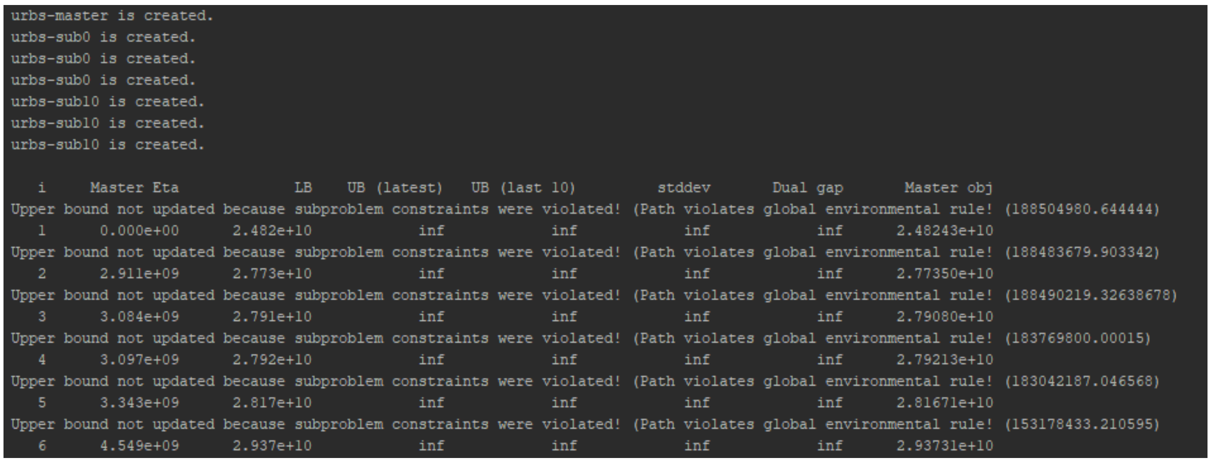


```
urbs-master is created.
urbs-subSouth is created.
urbs-subNorth is created.
urbs-subMid is created.

   i     omega    Master Eta     Sub Lambda    Lower Bound    Upper Bound       Dual gap     Master obj
   1       3      0.000e+00      2.470e+10           -inf            inf            inf      1.26000e+08
Cut skipped for subproblem South (Lambda = 0.0)
Cut skipped for subproblem North (Lambda = 0.0)
   2       3      2.470e+10      2.235e+04           -inf            inf            inf      2.48243e+10
Cut skipped for subproblem South (Lambda = 0.0)
Cut skipped for subproblem North (Lambda = 0.0)
   3       3      2.470e+10      1.829e+04           -inf            inf            inf      2.48259e+10
Cut skipped for subproblem South (Lambda = 0.0)
Cut skipped for subproblem North (Lambda = 0.0)
   4       3      2.473e+10      1.706e+04           -inf            inf            inf      2.48536e+10
Cut skipped for subproblem South (Lambda = 0.0)
Cut skipped for subproblem North (Lambda = 0.0)
   5       0      2.550e+10      7.267e+09      2.563e+10      3.289e+10      7.267e+09      2.56272e+10
```

This is the terminal output of Regional where the option `print_omega` is set to True. If this option is set, the sum of the sub problems `omega` variable is printed. You can see that it is set to zero every five iterations. Otherwise the output is equal to the output of Divide Timesteps.

Also you can see that the terminal output informs you if cuts are skipped for any sub problems which is a sign that it gets close to convergence.



```
urbs-master is created.
urbs-sub0 is created.
urbs-sub0 is created.
urbs-sub0 is created.
urbs-sub10 is created.
urbs-sub10 is created.
urbs-sub10 is created.

   i     Master Eta            LB    UB (latest)   UB (last 10)        stddev       Dual gap       Master obj
Upper bound not updated because subproblem constraints were violated! (Path violates global environmental rule! (188504980.644444)
   1      0.000e+00      2.482e+10           inf           inf           inf           inf      2.48243e+10
Upper bound not updated because subproblem constraints were violated! (Path violates global environmental rule! (188483679.903342)
   2      2.911e+09      2.773e+10           inf           inf           inf           inf      2.77350e+10
Upper bound not updated because subproblem constraints were violated! (Path violates global environmental rule! (188490219.32638678)
   3      3.084e+09      2.791e+10           inf           inf           inf           inf      2.79080e+10
Upper bound not updated because subproblem constraints were violated! (Path violates global environmental rule! (183769800.00015)
   4      3.097e+09      2.792e+10           inf           inf           inf           inf      2.79213e+10
Upper bound not updated because subproblem constraints were violated! (Path violates global environmental rule! (183042187.046568)
   5      3.343e+09      2.817e+10           inf           inf           inf           inf      2.81671e+10
Upper bound not updated because subproblem constraints were violated! (Path violates global environmental rule! (153178433.210595)
   6      4.549e+09      2.937e+10           inf           inf           inf           inf      2.93731e+10
```

Finally, this is the terminal output of SDDP, which is slightly different as it gives information about the average and the standard deviation of the last ten upper bounds which are relevant for the convergence of SDDP.

.h5 files

The .h5 files contain all information about the pyomo models except the equations. They are saved in the sub directory `h5_files` and they can be inspected in python using the function `urbs.load()`. Additionally one can choose to save .h5 files of intermediate steps every x iterations by using the option `save_h5_every_x_iterations`.

.lp files

If the option `save_lp_files` is set to True, the .lp files are saved in the sub directory `lp_files`. This feature is meant for debugging only, because it incurs a large overhead in terms of working memory and a smaller overhead in terms of run time. The .lp files, similar to the .h5 files, contain information about the model, but including the equations. They can be opened in a standard text editor or can directly be used by a solver (e.g. gurobi). Additionally one can choose to save .lp files of intermediate steps every x iterations by using the option `save_h5_every_x_iterations`.

Convergence Plot

If decomposition is done, the convergence of the upper and lower bound is shown in the file `bounds-scenario_name.png`. This plot is created with the function `plot_convergence()`.

Scenario Comparison Excel

The file `scenario-comparison.xlsx` contains concise information about the benders loop convergence for each scenario. The data for one scenario is appended to the excel with the function `append_df_to_excel()`.

Tracking file

If the option `save_hardware usage` is set to True, the file `scenario_name-tracking.txt` contains information about the memory and CPU percentage currently used and about the CPU time and real time used so far. This information is saved after solving the original problem and after each iteration of the benders loop. The tracking file is created with the method `create_tracking_file()` and updated with the method `update_tracking_file()`.

Log Files

The log file of the solver for each scenario is saved in the file `scenario_name.log`.

Plotting and Reporting

If the option plot_and_report is set to True, reporting (implemented in report.py) creates an excel output file and plotting (implemented in plot.py) a standard graph. Refer to the sections *Plotting* and *Reporting*.

Warning: Plotting and Reporting is so far only supported for the original problem (no decompositon method). If the option plot_and_report is True, the decomposition method is not None, and run_normal is True, Plotting and Reporting will be done for the normal (not decomposed) problem. If plot_and_report is True, the decomposition method is not None, and run_normal is False, the program will crash!

Benders Functions

The file benders.py contains two helper functions for the Benders loop:

- get_production_cost() calculates the cost of the capacity that needs to be installed additionally to the already installed capacities in the master problem to satisfy the maximal demand in all sub problems (used in Divide Timesteps and SDDP).
- convergence_check() updates the lower bound and the upper bound of the benders loop:

```
def convergence_check(master, subs, upper_bound, costs, decomposition_
↪method):
    """ Convergence Check

    Args:
        master: a Pyomo ConcreteModel Master instance
        subs: a Pyomo ConcreteModel Sub instances dict
        upper_bound: previously defined upper bound
        costs: extra costs calculated by get_production_cost()
        decomposition_method: The decomposition method which is used.␣
↪Must be in ['divide-timesteps', 'regional', 'sddp']

    Returns:
        GAP = Dual Gap of the Bender's Decomposition
        Zdo = Lower Bound
        Zup = Upper Bound

    Example:
        >>> upper_bound = float('Inf')
        >>> master_inst = create_model(data, range(1,25), type=2)
        >>> sub_inst = create_model(data, range(1,25), type=1)
        >>> costs = get_production_cost(...)
        >>> convergence_check(master_inst, sub_inst, Zup, costs)
    """
    lower_bound = master.obj()
```

First the lower bound is set to the current master objective. The optimal solution has to cost at least as much as the current objective for the following reasons:

  – The master problem objective consists of the costs of a part of the variables (the capacities) which it can optimize and a cost term given by the sub problems which is treated as constant.

- The cost term the master problem can optimize can only get higher in later iterations, because more constraints can be added to the master problem, but no constraints can be removed.
- The costs given by the sub problems can only get higher, because the bounds the sub problems receive from the master problem can only get tighter as the master problem acquires more cuts.

```
new_upper_bound = 0.0

for inst in subs:
    new_upper_bound += sum(subs[inst].costs[ct]() for ct in master.
↪cost_type)

if decomposition_method in ['divide-timesteps','sddp']:
    new_upper_bound += master.obj() - sum(master.eta[t]() for t in␣
↪master.tm) + costs
elif decomposition_method == 'regional':
    new_upper_bound += master.obj() - sum(master.eta[s]() for s in␣
↪master.sit) + costs
else:
    raise Exception('Invalid decomposition Method')
```

A solution is calculated for the current iteration in `new_upper_bound`. This solution is the sum of the sub problems costs, the costs (these are the costs of the capacity the master has to install to satisfy the maximum capacity needed by any sub problem) and the master objective minus the eta variables of the master objective which are the sub problem costs of the previous iteration.

```
upper_bound = min(upper_bound, new_upper_bound)
```

The upper bound is calculated by taking the current best solution (the minimum between the old best solution (`upper_bound`) and the new solution (`new_upper_bound`)). Obviously the best solution is at least as good as the best solution known so far.

```
gap = upper_bound - lower_bound

return gap, lower_bound, upper_bound
```

Update the gap and return the new values for lower bound, upper bound and gap.

Parallelization

The module `urbs.parallelization` allows to solve several sub problems in parallel using the python module `Pyro`. This section explains its main functions.

```
def start_pyro_servers(number_of_workers=None, verbose=False, run_
↪safe=True):
    """
    Starts all servers necessary to solve instances with Pyro. All servers␣
↪are started as daemons, s.t. if the main thread terminates or aborts,␣
↪the servers also shutdown.

    Args:
        number_of_workers: number of workers which are started. Default␣
↪value is the number of cores.
        verbose: If False output of the servers is suppressed. This is␣
↪usually desirable to avoid spamming the console window.
```

(continues on next page)

(continued from previous page)

```
        run_safe: If True a safety check is performed which ensures no␣
↪other program using pyro is running.

    Returns: list of processes which have been started so that they can␣
↪later be shut down again
    """
    from multiprocessing import Process
    # safety check to ensure no program using pyro is currently running
    if run_safe:
        pyro_safety_abort(run_safe=run_safe)
    # launch servers from code
    process_list = []
    # name server
    p = Process(target=start_name_server,kwargs={'verbose':verbose})
    p.daemon = True
    process_list.append(p)
    p.start()
    # dispatch server
    p = Process(target=start_dispatch_server,kwargs={'verbose':verbose})
    p.daemon = True
    process_list.append(p)
    p.start()
    # workers
    if number_of_workers is None:
        from multiprocessing import cpu_count
        number_of_workers = cpu_count()
    for i in range(0, number_of_workers):
        p = Process(target=start_pyro_mip_server,kwargs={'verbose':verbose}
↪)
        p.daemon = True
        process_list.append(p)
        p.start()
    # wait shortly to give servers time to start
    time.sleep(5)
    return process_list
```

The function `start_pyro_servers()` starts up all required servers (name sever, dispatch server and workers). It does this by creating a daemon process (process automatically terminates when main program terminates) for each server and starting it using the functions `start_name_server()`, `start_dispatch_server()` and `start_pyro_mip_server()`. It then returns a list of all processes started by these functions. All these functions are pretty simple and are not discussed in detail. With the parameter `number_of_workers` we can pass how many worker servers we desire. If it is not specified it is set to the number of cores by default. The option `verbose` is False by default, as it is usually desirable to keep the console clear of the servers output which makes the output a bit obscure. If the option `run_safe` is set to True, the function `pyro_safety_abort()` is run.

```
def pyro_safety_abort():
    """
    Check if there is a pyro name server running, which indicates that␣
↪another program using pyro might be running.
    This might lead to unexpected behaviour, unexpected shutdowns of some␣
↪of the servers or unexpected crashes in any of the programs.
    To avoid problems the program which called this function fails with an␣
↪Exception.
    """
```

(continues on next page)

(continued from previous page)

```
    import Pyro4
    try:
        Pyro4.locateNS()
    except:
        return
    raise Exception(
            'A Pyro4 name server is already running,'
            ' this indicates that other programs using Pyro are already␣
↪running,'
            ' which might lead to crashes in any of the programs.'
            ' To avoid this, this program is aborted.'
            ' If you want to run anyway, put run_safe to False and run␣
↪again.')
```

This function is a simple check if other programs are running which also use `Pyro` by checking if a pyro name server is already up. If another program is indeed using `Pyro` this could lead to unexpected behaviour or crashes. If you are sure no other program is using `Pyro`, but a name server is running anyway, you can either try to shutdown the nameserver or set the option `run_safe` to False. The last option is not recommended.

The function `solve_parallel()` needs to be called to solve several sub problems in parallel:

```
def solve_parallel(instances, solver, verbose=False):
    """
    Solves pyomo model instances in parallel using pyro

    Args:
        instances: instances dict
        solver: solver to be used for the problems
        verbose: If False output of the clients is suppressed. This is␣
↪usually desirable to avoid spamming the console window.

    Returns:
        A list of the solver results
    """
    if not verbose:
        # create a text trap and redirect stdout
        oldstdout = sys.stdout
        text_trap = io.StringIO()
        sys.stdout = text_trap

    from pyomo.opt.parallel import SolverManagerFactory

    solver_manager = SolverManagerFactory('pyro')
    if solver_manager is None:
        print("Failed to create solver manager.")
        sys.exit(1)

    action_handle_map = {}  # maps action handles to instances
    for i, inst in enumerate(instances):
        action_handle = solver_manager.queue(instances[inst], opt=solver,␣
↪tee=False)
        action_handle_map[action_handle] = "inst_{}".format(i)

    # retrieve the solutions
```

(continues on next page)

(continued from previous page)

```
    results = []
    for i in range(0, len(instances)):  # we know there are two instances
        this_action_handle = solver_manager.wait_any()
        results.append(solver_manager.get_results(this_action_handle))

    if not verbose:
        # now restore stdout function
        sys.stdout = oldstdout

    return results
```

The function works by setting up the `SolverManagerFactory` using `Pyro`. It then associates each instance with an action handle which it needs to retrieve the results after solving. The function returns the solved instances. The option `verbose` is set to False by default, because the output of the `SolverManagerFactory` is usually not relevant.

```
def shutdown_pyro_servers(process_list):
    """
    Terminates all processes in process_list

    Args:
        process_list: processes to be terminated
    """
    # shutdown servers
    for p in process_list:
        p.terminate()
```

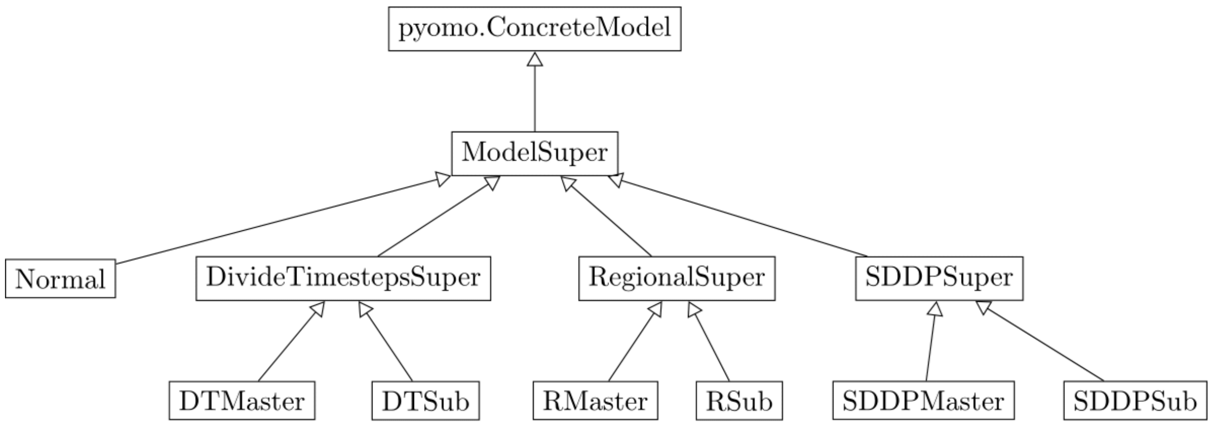
Finally the method `shutdown_pyro_servers()` shuts down the servers if given the process list returned by `start_pyro_servers()` as input.

Model Class Structure

This section explains how the models are grouped into different classes in the decomposition branch. It also highlights the differences between the decomposition models and the original model. To understand this section you should be familiar with the original model. If this does not apply to you, you should have a look at the sections *Overview* and perhaps *Mathematical Documentation* first.

Depending on which of the decomposition methods is used, the models look slightly different.

This is organized in a class structure as follows:



This graphic shows the classes and their inheritance. Classes in a lower level inherit from the classes in the level above if they are connected by an arrow.

- ModelSuper: Abstract class which unifies the model parts and constraint rules which are the same for all models. It inherits from `pyomo.ConcreteModel`.
- Normal: The urbs model if no decomposition method is used.
- DivideTimestepsSuper, RegionalSuper, SddpSuper: Abstract classes which contain the model parts which are the same for master and sub problems of the respective decomposition method.
- DivideTimestepsMaster, RegionalMaster, SddpMaster: The models for the master instance of the respective decomposition method.
- DivideTimestepsSub, RegionalSub, SddpSub: The models for the sub instances of the respective decomposition method.

It is possible to create instances of the master, sub and normal classes. Instances of the super classes make no sense by themselves and therefore the classes are abstract.

General Model Structure

For an overview over what kind of sets, variables, parameters and equations can be set in the model see *Overview* and *Mathematical Documentation* for more in depth explanations. Buy/Sell, Startup and demand site management features are not supported in the decomposition branch as of now. The modelling ideas which are common to all decomposition methods (and different or new compared to the normal model) are explained in the following sections. To understand what the specifics of each decomposition method are, key differences between them and the normal model are explained in the sections *Divide Timesteps Model*, *Regional Model* and *SDDP Model*.

Common concepts of decomposition methods

This section explains the parts of decomposition which are the same for all three methods, but different from the normal. In brief, all decomposition methods have a master problem which optimizes the overall costs. The master problem outsources some of its variables and therefore costs (`eta` or future costs) to the sub problems and imposes constraints on these problems. The goal of each sub problem is to minimize the violation of these constraints to eventually bring them down to zero. The constraint violation is given by the term `Lambda` times `omega`. This, at the same time, optimizes the costs of the sub problems, because one of the constraints given by the master problem is on the sub problems costs. If the sub problem cannot fulfill the constraints given by the master problem, it generates a cut for the master problem, which "informs" the master problem that the constraints are too tight. In the next iteration the master problem calculates a new solution with new constraints that consider the new information given by the cuts.

Sets, Variables, Parameters and Expressions

- Master objective: The master objective is to minimize the costs of the master problem, which are equal to the overall costs.
- Sub and master costs: The costs of the sub problems and the master problem are slightly different for each decomposition method, depending on which costs incur in which problem.

- `eta`/future costs: The variable `eta` in the master describes the future cost for each sub problem (the cost that the subproblem is expected to have). The sub problems have a variable `eta_res` (restriction on eta) which is equal to the variable `eta` from the same iteration in the master. SDDP is an exception here, because the master problem can only communicate with the first sub problem, so every sub problem has its own `eta` variable which sets the restriction `eta_res` for the next sub problem.
- `Lambda`: The master problem imposes constraints on the sub problems. The `Lambda` variable of the sub problem is the maximal violation of any such constraints. The sub problems objective is to minimize `Lambda`. If `Lambda` is zero, this means that the sub problem does not violate any constraints from the master problem. This also means that the sub problem does not contribute a new cut to the master problem. If `Lambda` is zero for all sub problems, no further cuts are added to the master problem, so the feasible region of the master problem is known and an optimal solution can be found.
- `omega`: $(\omega, \omega_0)^T$ is called the cost vector. The constraint on the costs of the sub problem may be violated by no more than `omegazero` times `Lambda`, while a constraint on any other variable may be violated by no more than `omega` times `Lambda`. This approach allows the sub problems to generate cuts which are close (facet cuts) to the feasible solution area of the master problem and thus lead to relatively fast convergence. For Divide Timesteps and SDDP we fix both `omega` and `omegazero` to one which leads to promising results. For regional we also choose both to be one, but every five iterations, we set `omega` to zero. This has the effect that the sub problems are forced to not make relaxing assumptions which in turn leads to the sub problems using the expensive slack power plants and not assumed transmissions from other sites. This results in a faster estimation of an upper bound.
- `dual`: If `dual` is true, the dual variable of the pyomo model is set such that the dual variables are saved.
- Capacity rules and parameters: The capacity constraints have a special form: $capacity <= cap_{new} + cap_{installed} + cap_{relax}$. The advantage of these constraints are that they can be used in the normal, master and sub models by setting the involved parameters and expressions correctly. E.g. for the normal model `cap_relax` is zero, so that the capacity is equal to the installed capacity plus the new capacity. To be more exact this holds for all models which are allowed to expand the capacities. These are the normal model as well as all master models and the regional sub models. In sub problems of Divide Timesteps and SDDP on the other hand, the new capacity and the installed capacity are set to the values of the master problem as only the master problem is allowed to expand capacities. The parameter `cap_relax` though is set to `omega` times `Lambda`, because the sub problems are allowed to violate constraints by this amount.

Rules

- Objective Rules: The objective rules are the same for all decomposition methods. The master optimizes its cost and the sub problem optimize their `Lambda` variable.
- `def_capacity_rules`: Explained in detail in *Sets, Variables, Parameters and Expressions*
- `def_capacity_l_rule`: The lower capacity rule is used in Divide Timesteps and SDDP and forces sub problems to have at least a certain amount of capacity in the beginning. Although this rule seems not intuitive, it is necessary, because even if the sub problem does not need the capacity for itself it still needs to pay the running costs if it is installed.

Functions

- Cut generation: The sub problems generate cuts for the master problem (Divide Timesteps, Regional) or the previous problem (SDDP). Cut generation is different for each of the methods.
- Set boundaries: The method `set_boundaries()` is used to set a restriction variable in the sub problem (e.g. `eta_res`) to the corresponding value in the master problem (e.g. `eta`).
- Wrapper methods: There are a couple of methods that provide a wrapper for the underlying `pyomo.ConcreteModel`: `get_attribute()`, `get_attribute_at()`, `get_cost()` and `get_duals()` which are pretty self explanatory from the doc strings. There's also a method `solve()` which can be called in the way `model.solve(solver)`.

Divide Timesteps Model

This part of the documentation explains the parts of the Divide Timesteps model which are different to the normal model.

Sets, Variables, Parameters and Expressions

- `support_timesteps`: The support time steps are defined as a set in both the master and the sub problems, and give the time steps at which the original problem is split into sub problems.
- `e_co_stock_res` defines the restrictions on the stock commodity for the sub problems.

Rules

- `def_costs_rule`:
  - The costs of the master problem are the investment costs and fix costs for all capacity variables plus the sum of costs of all sub problems, which are stored in `FutureCosts`.
  - The costs of the sub problem consists of the three cost types not accounted for in the master problem. These are variable costs, which are costs varying with the usage of commodities, fuel costs, which depend on the use of stock commodities, and environmental costs, which depend on the use of taxed environmental commodities. Also compare with *Cost Variables* (though not all cost variables of the master branch are supported yet).
- `res_storage_state_by_capacity_rule`: This rule is the same as in the normal, except that the constraints for the support steps in the sub problems are skipped, because they are also included in the master problem and the constraint is enforced there.
- `res_co2_generation_rule`/`res_global_co2_limit_rule`: Make sure that the master and the sub problems respectively don't pass the global CO2 limit.
- `sub_costs_rule`: Assures that the costs of the sub problems cannot be higher than the restriction on costs given by the master problem plus `omega` times `Lambda`.
- `sub_commodity_source`: This rule enforces that the sub problems cannot use more of a stock commodity than allowed by the restriction `e_co_stock_res` plus the relaxing expression `omega` times `Lambda`.

Functions

Cut Generation

This section explains the function `add_cut()` in the Divide Timesteps Master in detail.

```
def add_cut(self, cut_generating_problem, readable_cuts=False):
    """
    Adds a cut to the master problem, which is generated by a sub problem

    Args:
        cut_generating_problem: sub problem which generates the cut
        readable_cuts:  scale cuts to make them easier to read (may cause␣
↪numerical issues)
    """
    if cut_generating_problem.Lambda() < 0.000001:
        print('Cut skipped for subproblem ' + str(cut_generating_problem)␣
↪+ ' (Lambda = ' + str(
            cut_generating_problem.Lambda()) + ')')
        return
```

First, check if `Lambda` is very close to zero. If `Lambda` is zero, this means that the sub problem does not violate any constraints passed to it by the master problem. This in turn means that the sub problem yields a feasible solution and does not add a new constraint to the master problem. In this case we don't add a cut and simply return.

```
# dual variables
multi_index = pd.MultiIndex.from_tuples([(t,) + sto
                                        for t in self.t
                                        for sto in self.sto_tuples],
                                       names=['t', 'sit', 'sto', 'com'])
dual_sto = pd.Series(0, index=multi_index)
dual_sto_help = get_entity(cut_generating_problem, 'res_initial_and_final_
↪storage_state')
dual_sto = dual_sto.add(-abs(dual_sto_help.loc[[cut_generating_problem.
↪ts[1]]]), fill_value=0)
dual_sto = dual_sto.add(abs(dual_sto_help.loc[[cut_generating_problem.ts[-
↪1]]]), fill_value=0)
```

Next, we initialize the dual variables. For every constraint the corresponding dual variable states how much the objective would change if the constraint is changed by one. Note that this means the duals are not really variables (in the mathematical sense), but rather fixed rational numbers. The storage constraint dual is made negative for the first time step of the cut generating (sub) problem, because increasing the storage available in the beginning would decrease the objective function. Similar the dual is made positive for the last time step of the cut generating problem, because increasing the storage which needs to be left at the end of the cut generating problem would increase the objective function.

```
dual_pro = get_entity(cut_generating_problem, 'def_process_capacity')
dual_tra = get_entity(cut_generating_problem, 'def_transmission_capacity')
dual_sto_cap = get_entity(cut_generating_problem, 'def_storage_capacity')
dual_sto_capl = get_entity(cut_generating_problem, 'def_storage_capacity_l
↪')
dual_sto_pow = get_entity(cut_generating_problem, 'def_storage_power')
dual_com_src = get_entity(cut_generating_problem, 'sub_commodity_source')
dual_env = get_entity(cut_generating_problem, 'res_global_co2_limit')
```

(continues on next page)

(continued from previous page)

```
dual_zero = cut_generating_problem.dual[cut_generating_problem.sub_costs]
Lambda = cut_generating_problem.Lambda()
```

Next, we initialize all other dual variables. For every constraint there is exactly one dual. Note that one rule can describe more than one constraint and in turn the corresponding dual variable is actually a vector of dual variables. As an example consider `def_process_capacity`. This rule defines a constraint for each process which means `dual_pro` contains one dual variable for every one of these constraints. In Divide Timesteps there are the capacity constraints, the commodity constraint (`sub_commodity_source`), the CO2 constraint (`res_global_co2_limit`) and the cost constraint. To generate the cut we also need the value of `Lambda` for the cut generating problem.

```
cut_expression = - 1 * (sum(dual_pro[pro] * self.cap_pro[pro] for pro in␣
↪self.pro_tuples) +
                                sum(dual_tra[tra] * self.cap_tra[tra] for␣
↪tra in self.tra_tuples) +
                                sum((dual_sto_cap[sto] - dual_sto_
↪capl[sto]) * self.cap_sto_c[sto] for sto in self.sto_tuples) +
                                sum(dual_sto_pow[sto] * self.cap_sto_
↪p[sto] for sto in self.sto_tuples) +
                                sum([dual_sto[(t,) + sto] * self.e_sto_
↪con[(t,) + sto]
                                     for t in self.t
                                     for sto in self.sto_tuples]) +
                                sum([dual_com_src[com] * self.e_co_
↪stock[(cut_generating_problem.tm[-1],) + com]
                                     for com in self.com_tuples if
                                     com[1] in self.com_stock
                                     and not math.isinf(self.commodity.
↪loc[com]['max'])]) +
                                sum([dual_env[0] * self.e_co_stock[(cut_
↪generating_problem.tm[-1],) + com]
                                     for com in self.com_tuples
                                     if com[1] in self.com_env]) +
                                dual_zero * self.eta[cut_generating_
↪problem.tm[-1]])
```

With the dual variables we can generate the cut expression: The cut expression is the sum of all dual variables times the corresponding variables in the master instance. This reflects that by increasing one variable in the master instance (e.g. a process: `cap_pro[pro]`) the objective function of the sub problem would change by the corresponding dual (e.g. `dual_pro[pro]`). As increasing the capacity would decrease the objective function and decreasing it would increase the objective function we have to multiply by minus one. The same holds for the constraints on commodities, CO2 and costs (allowing for more commodities/CO2/costs, decreases the objective function).

```
# cut generation
if readable_cuts and dual_zero != 0:
    cut = 1 / (-dual_zero) * cut_expression >= 1 / (-dual_zero) * (Lambda␣
↪+ cut_expression())
else:
    cut = cut_expression >= Lambda + cut_expression()
self.Cut_Defn.add(cut)
```

The cut expression can be evaluated (with `cut_expression()`) for the current variables in the master problem. We know that using the current values of the master variables the sub problem cannot be solved without violating at least one constraint by `Lambda` (because the sub problem minimizes `Lambda`). This implies that in future iterations the cut expression has to be at least the evaluated cut expression plus `Lambda` for the sub problem to become feasible (`Lambda` is (almost) zero). This is the cut we add to the master problem.

If `readable_cuts` is True we multiply both sides by one divided through minus `dual_zero`, which corresponds to down scaling both sides with the negative of the dual of the sub problem costs. This gives a different representation of the cuts which is helpful to their mathematical interpretation. On the other hand it can lead to numerical problems, because a multiplication with a very small number could happen,so the feature is turned off by default.

Regional Model

This part of the documentation explains the parts of the regional model which are different to the normal model. A special case within the regional model is a sub problem with its own specified file. This sub problem then has its own sub sub problems for which it can set restrictions. Also the sub problem with input file has to handle the problem of managing its transmissions, because the master problem is oblivious to the different regions.

Modelling a region with input file

When modelling one or even several of the regions with their own input file, the transmission efficiencies e_{af} have to be set carefully to avoid discrepancies in the model. The source of discrepancies is that the master model M has an efficiency between two sub regions A and B. Now, if A and/or B have their own input file, they can assign efficiencies between their sub sites (a1, a2, b1, b2,. . .) and other regions (A,B,C,. . .) as well. When we look at one single transmission line, arithmetic operations with the efficiency happens at three points (also compare with *Rules*). The variables `e_tra_in` and `e_tra_out` are abbreviated with their mathematical symbols $\pi_{aft}^{\text{model,in}}$ and $\pi_{aft}^{\text{model,out}}$ (compare *Mathematical Documentation*) in the following.

1. In the region the transmission line starts (lets say A), the `res_export_rule` (or `sub_e_tra_rule` if A does not have its own input file) uses `e_tra_out` which is an implicit multiplication with the efficiency e_{af}^{A} given in A (by the `transmission_output_rule`) as:

$$\pi_{aft}^{\text{A,out}} = \pi_{aft}^{\text{A,in}} \cdot e_{af}^{\text{A}}$$

2. The very same rule compares `e_tra_out` with `e_tra_out_res` ($\pi_{aft}^{\text{A,out,res}}$) which implies a division through the efficiency given in the master problem e_{af}^{M}, because `e_tra_out_res` is passed by the master problem, where it is calculated from `e_tra_in`. We can consider this as a division, because:

$$\pi_{aft}^{\text{A,out}} \geq \pi_{aft}^{\text{A,out,res}} + \lambda\omega$$
$$\pi_{aft}^{\text{A,out}} \geq \pi_{aft}^{\text{M,out}} + \lambda\omega$$
$$\pi_{aft}^{\text{A,out}} \geq \pi_{aft}^{\text{M,in}} \cdot e_{af}^{\text{M}} + \lambda\omega$$
$$\frac{\pi_{aft}^{\text{A,out}}}{e_{af}^{\text{M}}} \geq \pi_{aft}^{\text{M,in}} + \frac{\lambda\omega}{e_{af}^{\text{M}}}$$

3. Finally at the end of the transmission line (in B), a multiplication with the efficiency e_{af}^{B} given in B happens by the `transmission_output_rule`:

$$\pi_{aft}^{\text{B,out}} = \pi_{aft}^{\text{B,in}} \cdot e_{af}^{\text{B}}$$

and further, because `e_tra_in` in B comes from `e_tra_in_res` in B which is passed by the master considering the `res_import_rule` (or `sub_e_tra_rule` if B does not have its own input file):

$$\begin{aligned}
\pi_{aft}^{\text{B,out}} &= \pi_{aft}^{\text{B,in}} \cdot e_{af}^{\text{B}} \\
\pi_{aft}^{\text{B,out}} &\leq (\pi_{aft}^{\text{B,in,res}} + \lambda\omega) \cdot e_{af}^{\text{B}} \\
\pi_{aft}^{\text{B,out}} &\leq (\pi_{aft}^{\text{M,in}} + \lambda\omega) \cdot e_{af}^{\text{B}}
\end{aligned}$$

Considering the equations after convergence of the benders loop, `Lambda` is zero and the inequalities are equalities. Combining the equations from 1. and 2. and 3. gives:

$$\pi_{aft}^{\text{B,out}} = \pi_{aft}^{\text{A,in}} \cdot \frac{e_{af}^{\text{A}} \cdot e_{af}^{\text{B}}}{e_{af}^{\text{M}}}$$

When modeling with sub input files this equation should be kept in mind. Also keep in mind that the sub problems without input file have the same efficiency as the master problem and the fraction can be reduced.

Sets, Variables, Parameters and Expressions

- `e_co_stock_res` defines the restrictions on the stock commodity for the sub problems. In regional this is only relevant for the CO2 restriction.
- `e_tra_in_res` defines the restrictions on incoming transmissions for the sub problems.
- `e_tra_out_res` defines the restrictions on outgoing transmissions for the sub problems.
- `hvac`: This parameter gives the current capacity of ingoing transmissions from one site to another. It is needed for the `res_hvac_rule` for sub problems with input files (see *Rules*).
- Sub with input file:
  - `e_import_res`, `e_export_res`: The restrictions on import and export for the sub problem.
  - `cap_tra`, `cap_tra_new`: Like the master problem the sub problem needs to be able to install transmission lines between its own sub problems.

Rules

- `def_costs_rule`
  - Master: The costs of the master problem consist of the transmission costs (Investment, Fixed and Variable) and the sum of the sub problems costs, which are stored in `FutureCosts`.
  - Sub without file: The sub problems cost consist of all costs except transmission within its site. This includes Investment, Fixed, Variable, Fuel and Environmental costs.
  - Sub with file: If the sub has a specified input file it has the same costs as a sub problem without input file, but in addition it has the Investment, Fixed and Variable costs for transmissions between its own sub sites.
- `sub_costs_rule`: Assures that the costs of the sub problem cannot be higher than the restriction on costs given by the master problem plus `omega` times `Lambda`.

- `res_global_co2_limit_rule`:
  - Master problem: Makes sure that global CO2 limit is not violated.
  - Sub problems: Assure that sub problems can only violate their CO2 restriction given by the master by at most `omega` times `Lambda`
- `hvac_rule`: Initializes the parameter hvac.
- Sub without file only:
  - `sub_e_tra_rule`: Assures that the sub problem can not import more than the restriction given by the master problem plus `omega` times `Lambda`. Also assures that the problem has to export at least as much as given by the master problem minus `omega` times `Lambda`.
- Sub with file only:
  - `res_hvac_rule`: Makes sure that the sum of transmission capacities going out from the sub sites of the current sub problem C to another sub problems site S are not more than the transmission capacity between C and S in the master problem plus `omega` times `Lambda`.
  - `res_export_rule`, `res_import_rule`: Similar to `res_hvac_rule`, these rules make sure that the sum of export/import from the sub sites of the current sub problem C to another sub problem site S match the export/import between C and S determined in the master problem. They are allowed to vary by a factor of `omega` times `Lambda`.

Functions

Cut Generation

This section explains the function `add_cut()` in the Regional Master in detail.

```
def add_cut(self, cut_generating_problem, sub_in_input_files):
    """Adds a cut, which is generated by a subproblem, to the master␣
↪problem

    Args:
        cut_generating_problem: sub problem instance which generates the␣
↪cut
        sub_in_input_files: If true, the cut generating problem is in the␣
↪list of filenames to Excel spread sheets for sub regions
    """
    if cut_generating_problem.Lambda() < 0.000001:
        print('Cut skipped for subproblem ' + cut_generating_problem.sub_
↪site[1] +
              ' (Lambda = ' + str(cut_generating_problem.Lambda()) + ')')
        return
```

First, check if `Lambda` is very close to zero. If `Lambda` is zero, this means that the sub problem does not violate any constraints passed to it by the master problem. This in turn means that the sub problem yields a feasible solution and does not add a new constraint to the master problem. In this case we don't add a cut and simply return.

```
# subproblem with input file
if sub_in_input_files:
    # dual variables
```

(continues on next page)

(continued from previous page)

```
    dual_imp = get_entity(cut_generating_problem, 'res_import')
    dual_exp = get_entity(cut_generating_problem, 'res_export')
    dual_cap = get_entity(cut_generating_problem, 'res_hvac')
    dual_env = get_entity(cut_generating_problem, 'res_global_co2_limit')
    dual_zero = cut_generating_problem.dual[cut_generating_problem.sub_
↪costs]
    Lambda = cut_generating_problem.Lambda()
```

The cuts look different depending on whether the cut generating problem has its own input file. First, we look at the case of the problem having its own input file. We initialize the dual variables, which say how much the objective function changes when a constraint changes. For every constraint there is exactly one dual. Note that one rule can describe more than one constraint and in turn the corresponding dual variable is actually a vector of dual variables. As an example consider `res_import`. This rule defines a constraint for each transmission line which means `dual_imp` contains one dual variable for every one of these constraints. In the case of a sub problem with its own input file there are constraints on the import, export, transmission capacity (`res_hvac`), CO2 and the costs. We also need the sub problems variable `Lambda`.

```
cut_expression = - 1 * (sum([dual_imp[tm, tra[0]] * self.e_tra_in[(tm,) +␣
↪tra]
                                 for tm in self.tm
                                 for tra in self.tra_tuples
                                 if tra[1] == cut_generating_problem.sub_
↪site[1]]) -
                            sum([dual_exp[tm, tra[1]] * self.e_tra_out[(tm,
↪) + tra]
                                 for tm in self.tm
                                 for tra in self.tra_tuples
                                 if tra[0] == cut_generating_problem.sub_
↪site[1]]) +
                            sum([dual_cap[tra[0]] * self.cap_tra[tra]
                                 for tra in self.tra_tuples
                                 if tra[1] == cut_generating_problem.sub_
↪site[1]]) +
                            sum([dual_env[0] * self.e_co_stock[(tm,) + com]
                                 for tm in self.tm
                                 for com in self.com_tuples
                                 if com[0] == cut_generating_problem.sub_
↪site[1] and com[1] in self.com_env]) +
                            dual_zero * self.eta[cut_generating_problem.
↪sub_site[1]])
```

With the dual variables we can generate the cut expression: The cut expression is the sum of all dual variables times the corresponding variables in the master instance. This reflects that by increasing one variable in the master instance (e.g. the incoming transmission at a timestep: `e_tra_in[(tm,) + tra]`) the objective function of the sub problem would change by the corresponding dual (e.g. `[dual_imp[tm, tra[0]]`). As increasing the incoming transmission would decrease the objective function and decreasing it would increase the objective function we have to multiply by minus one. The same holds for the constraints on transmission capacity, CO2 and costs. On the other hand if we increase export, the objective function increases, hence the minus before the sum over all exports.

```
else:
    # dual variables
    dual_tra = get_entity(cut_generating_problem, 'sub_e_tra')
```

(continues on next page)

(continued from previous page)

```
    dual_env = get_entity(cut_generating_problem, 'res_global_co2_limit')
    dual_zero = cut_generating_problem.dual[cut_generating_problem.sub_
↪costs]
    Lambda = cut_generating_problem.Lambda()
```

If the cut generating sub problem has no input file, we only have constraints on transmissions (in- and outgoing transmissions are both in the rule `sub_e_tra`), CO2 and costs.

```
# cut generation
cut_expression = - 1 * (sum([dual_tra[(tm,) + tra] * self.e_tra_in[(tm,) +␣
↪tra]
                             for tm in cut_generating_problem.tm
                             for tra in cut_generating_problem.tra_tuples
                             if tra[1] in cut_generating_problem.sub_
↪site]) -
                        sum([dual_tra[(tm,) + tra] * self.e_tra_out[(tm,)␣
↪+ tra]
                             for tm in cut_generating_problem.tm
                             for tra in cut_generating_problem.tra_tuples
                             if tra[0] in cut_generating_problem.sub_
↪site]) +
                        sum([dual_env[0] * self.e_co_stock[(tm,) + com]
                             for tm in cut_generating_problem.tm
                             for com in cut_generating_problem.com_tuples
                             if com[1] in cut_generating_problem.com_env])␣
↪+
                        dual_zero * self.eta[cut_generating_problem.sub_
↪site[1]])
```

Like before, we use this to generate the cut expression. Note that `e_tra_in` is split into import and export, where import needs to be multiplied by minus one, while export is not.

```
cut = cut_expression >= Lambda + cut_expression()
self.Cut_Defn.add(cut)
```

The cut expression can be evaluated (with `cut_expression()`) for the current variables in the master problem. We know that using the current values of the master variables the sub problem cannot be solved without violating at least one constraint by `Lambda` (because the sub problem minimizes `Lambda`). This implies that in future iterations the cut expression has to be at least the evaluated cut expression plus `Lambda` for the sub problem to become feasible (`Lambda` is (almost) zero). This is the cut we add to the master problem.

SDDP Model

This model explains the differences between the SDDP model and the normal model and also emphasizes key differences to the Divide Timesteps model which is very similar to SDDP.

Sets, Variables, Parameters and Expressions

- `support_timesteps`: Determine at which time steps the original problem is split into sub problems.

- `com_max_tuples`: A set of all stock and environmental variables which have a maximum allowed usage amount.
- `e_co_stock_state`: This variable gives the usage of a stock commodity up to a time step.
- `e_co_stock_state_res`: This variable is a constraint on the state of a stock commodity at the beginning of a sub problem given by the previous problem.

Rules

There are some additional or different rules in SDDP compared to Divide Timesteps. These rules are there, to ensure the sub problems combined do not cross global restrictions on stock or CO2 , or other constraints that cannot be enforced in the master problem like in Divide Timesteps where the master has access to all support steps, but rather must be passed from sub problem to sub problem.

- `def_costs_rule`:
  - Master: The master costs includes the investment and fixed costs for all capacity variables, which can only be expanded in the master problem. If the first support step is not equal to the first time step, the master problem also has to carry the variable costs, fuel costs and environmental costs which occur in the time steps before the first support step. The cost of the first sub instance is added in the future costs (this means the master also includes all sub problem costs, because the first subproblems costs includes the costs of the second, the costs of the second the costs of the third and so on).
  - Subs: The costs of the sub problem consists of the three time dependent cost types. These are variable costs, which are costs varying with the usage of commodities, fuel costs, which depend on the use of stock commodities, and environmental costs, which depend on the use of taxed environmental commodities. Also compare with *Cost Variables* (though not all cost variables of the master branch are supported yet). Additionally it contains the cost of the next sub problem in its future costs.
- `res_storage_state_by_capacity_rule`: Like in the original problem, except that in the sub problems the constraint need not be enforced for the first time step, because the first timestep is set by the previous problem.
- `res_initial_storage_state_rule`: Unlike the rule `res_initial_and_final_storage_state_rule` in Divide Timesteps this rule is only included in the master instance and makes sure that the initial storage state is correct.
- `final_storage_state_rule`: This rule makes sure that the final storage state is correct.
- `sub_storage_content_rule`: This rules assures that the storage content in the first timestep of a sub problem obeys the storage content restriction given by the previous problem up to a deviation of `omega` times `Lambda`.
- `sub_com_generation_rule`: This rule asserts that the stock state (`e_co_stock_state`, the amount of stock used so far) is at least the stock state restriction minus `omega` times `Lambda`.
- `com_total_rule`: Asserts that the Env/Stock generation per site limitation is obeyed.
- `com_state_rule`: This rule asserts that the stock state in time step t is equal to the stock state in time step t-1 plus the stock used in timestep t.
- `global_co2_limit_rule`: Asserts that the global CO2 limit is not exceeded.

- `sub_costs_rule`: Assures that the costs of the sub problem cannot be higher than the restriction on costs given by the master problem plus `omega` times `Lambda`.

Functions

Cut Generation

There are two methods in SDDP for cut generation:

- `add_cut()` calculates the weighted cut between the cuts of the possible realizations.

```
def add_cut(self, realizations, cut_generating_problems, current_
↪realized, probabilities):
    """
    Adds a cut to this problem (in Sddp cuts can be added to both master␣
↪and sub problems)

    Args:
        realizations: possible realizations (e.g. "low", "mid", "high")␣
↪of the following supportsteps problem (= cut generating problems)
        cut_generating_problems: the realizations of the sub problem in␣
↪the next timestep which generate the cut
        current_realized: realized instance of current problem
        probabilities: probabilities of realizations
    """
    cur_probs = {}
    for cur_real in realizations:
        if cut_generating_problems[cur_real].Lambda() > 0.0000001:
            cur_probs[cur_real] = cut_generating_problems[cur_real]
        else:
            print('Cut skipped for subproblem ' + '(' + str(cut_
↪generating_problems[cur_real].ts[1]) + ', ' + cur_real +
                  '), Lambda = ' + str(cut_generating_problems[cur_real].
↪Lambda()))
```

First, we check if `Lambda` is very close to zero for any cut generating problem. If `Lambda` is zero, this means that the realization of the sub problem does not violate any constraints passed to it by the previous problem. This in turn means that the realization yields a feasible solution and does not contribute to the weighted cut for the previous problem.

```
if len(cur_probs) > 0:
    self.Cut_Defn.add(
        sum(probabilities[cur_real] * self.get_cut_expression(cur_
↪probs[cur_real])
            for cur_real in cur_probs)
        >= sum(probabilities[cur_real] *
               (cur_probs[cur_real].Lambda() + current_realized.get_
↪cut_expression(cur_probs[cur_real])())
               for cur_real in cur_probs))
```

If there is at least one cut which has not been skipped, we generate the weighted cut for the current problem. To obtain one cut we take the cut expression generated by `get_cut_expression()` for each possible realization of the next timestep. We know that using the current values of the current problems variables the problem in the next time step cannot be solved without violating at least one constraint by `Lambda` (because the sub problem minimizes `Lambda`). This implies that in future iterations the cut expression has to be at least the evaluated cut expression plus `Lambda` for the sub problem to become feasible (`Lambda` is (almost) zero). Because we can only evaluate the cut expression for the realized instance (we only know the values for the variables of the instance we solved in the forward recursion), we use its cut expression as an approximate substitute for all the realizations. To obtain the weighted cut we multiply each generated cut with the realization's probability on both sides and take their sum.

- `get_cut_expression()` creates the cut expression for the current realization generated by one possible realization in the next time step.

```
def get_cut_expression(self, cut_generating_problem):
    """
    Calculates the cut expression for one realization

    Args:
        cut_generating problem: the realization which generates the␣
↪cut

    Returns:
        the generated cut expression
    """
    multi_index = pd.MultiIndex.from_tuples([(t,) + sto
                                             for t in cut_generating_
↪problem.t
                                             for sto in cut_
↪generating_problem.sto_tuples],
                                            names=['t', 'sit', 'sto',
↪'com'])
    dual_sto = pd.Series(0, index=multi_index)

    dual_sto_help = get_entity(cut_generating_problem, 'sub_storage_
↪content')
    dual_sto = dual_sto.add(-abs(dual_sto_help.loc[[cut_generating_
↪problem.ts[1]]]), fill_value=0)
```

We start with initializing the dual variables. For every constraint the corresponding dual variables states how much the objective would change if the constraint is changed by one. Note that this means the duals are not really variables (in the mathematical sense), but rather fixed rational numbers. The storage constraint dual is made negative for the first time step of the cut generating problem, because increasing the storage available in the beginning would decrease the objective function. Unlike Divide Timesteps there is no constraint on the last time step of a sub problem, because the master problem has no access to that time step.

```
dual_pro = get_entity(cut_generating_problem, 'def_process_capacity')
dual_tra = get_entity(cut_generating_problem, 'def_transmission_
↪capacity')
dual_sto_cap = get_entity(cut_generating_problem, 'def_storage_
↪capacity')
dual_sto_capl = get_entity(cut_generating_problem, 'def_storage_
↪capacity_l')
dual_sto_pow = get_entity(cut_generating_problem, 'def_storage_power')
dual_com = get_entity(cut_generating_problem, 'sub_com_generation')
dual_zero = cut_generating_problem.dual[cut_generating_problem.sub_
↪costs]
```

Next, we initialize all other dual variables. For every constraint there is exactly one dual. Note that one rule can describe more than one constraint and in turn the corresponding dual variable is actually a vector of dual variables. As an example consider `def_process_capacity`. This rule defines a constraint for each process which means `dual_pro` contains one dual variable for every one of these constraints. In SDDP there are the capacity constraints, the generation constraint (`sub_com_generation`), which unifies the commodity and environmental constraints, and the cost constraint. To generate the cut we also need the value of `Lambda` for the cut generating problem.

```
cut_expression = - 1 * (sum(dual_pro[pro] * self.cap_pro[pro]
                            for pro in self.pro_tuples) +
                        sum(dual_tra[tra] * self.cap_tra[tra]
                            for tra in self.tra_tuples) +
                        sum((dual_sto_cap[sto] - dual_sto_capl[sto]) * self.
↪cap_sto_c[sto]
                            for sto in self.sto_tuples) +
                        sum(dual_sto_pow[sto] * self.cap_sto_p[sto]
                            for sto in self.sto_tuples) +
                        dual_zero * self.eta)

cut_expression += -1 * (sum([dual_sto[(self.t[-1],) + sto] * self.e_
↪sto_con[(self.t[-1],) + sto]
                             for sto in self.sto_tuples]) -
                        sum([dual_com[(self.t[-1],) + com] * self.e_co_
↪stock_state[
                            (self.t[-1],) + com]
                             for com in self.com_tuples if com in self.com_
↪max_tuples])
                        )
```

With the dual variables we can generate the cut expression: The cut expression is the sum of all dual variables times the corresponding variables in the current instance. This reflects that by increasing one variable in the current instance (e.g. a process: `cap_pro[pro]`) the objective function of the sub problem would change by the corresponding dual (e.g. `dual_pro[pro]`). As increasing the capacity would decrease the objective function and decreasing it would increase the objective function we have to multiply by minus one. The same holds for the cost constraint, while the generation constraint is not multiplied by minus one (or to be more precise in the implementation it is subtracted and then multiplied by minus one, which is equivalent). This makes sense, because the generation constraint says how much of the commodity has already been generated in the case of CO2 or used in the case of stock commodities. If the amount of CO2 generated or stock commodities used increases the objective function increases.

```
return cut_expression
```

Return the generated cut expression.

Create Uncertainty

To introduce uncertainty in the data we use the function `create_uncertainty_data()` which itself uses the function `create_uncertainty_supim()`.

```
def create_uncertainty_data(self, data, factor):
    """
    Change dataframe to include modified uncertain time series
```

(continues on next page)

(continued from previous page)

```
    Args:
        data: pandas DataFrame with original data
        factor: float, between -1 and 1, which corresponds to the␣
↪realization of the uncertainty

    Returns:
        pandas DataFrame with modified data
    """

    # get supim sheet
    supim = data['supim']
    new_data = data.copy()
    new_supim = supim.copy(deep=True)
    wind_supim = new_supim.xs('Wind', axis=1, level=1)
    help_df = self.create_uncertainty_supim(wind_supim, factor)
    help_df.columns = pd.MultiIndex.from_product([help_df.columns, ['Wind
↪']])
    new_supim.loc[:, (slice(None), 'Wind')] = help_df
    new_data['supim'] = new_supim

    return new_data
```

The uncertainty data is created by copying the old data, then introducing uncertainty using the function `create_uncertainty_supim()` for all desired supim time series (in this case only done for wind). The in this way newly created supim data is inserted back into the data. How much uncertainty is introduced is controlled by the passed factor and is passed on to `create_uncertainty_supim()`.

```
def create_uncertainty_supim(self, supim, factor):
    """
    create convex combination of supim time series for different scenarios

    Args:
        supim: pandas Series or DataFrame of supim time series of a␣
↪specific commodity
        factor: float, between -1 and 1, which corresponds to the␣
↪realization of the uncertainty

    Returns:
        pandas Series or DataFrame with convex combination
    """
    if factor < 0:
        supim_convex = (1 - abs(factor)) * supim
    elif factor > 0:
        supim_convex = abs(factor) + (1 - abs(factor)) * supim
    else:
        supim_convex = supim

    return supim_convex
```

This function manipulates a supim time series by taking a convex combination of the minimum or maximum possible value depending on whether factor is negative or positive respectively. The minimum value for any supim series is 0 and the maximum value is 1. The value of the factor is fixed for the entire time series.

Developers Guide

This guide makes suggestions on how to improve, use, and extend the code as well as how to unify it with the urbs master branch.

Improving the code

- Create dedicated SDDP scenarios: Currently the supim data series that is changed in the SDDP branch is hardcoded to be wind in `create_uncertainty_data` in the file `sddp_sub`. Likewise the possible realizations and their probabilities are hardcoded in `runme.py`. It would be nice to be able to create a SDDP scenario which defines which supim series is modified and then to define realizations and their probabilities. In this way it should be possible to vary several supim series at once (e.g. you could then have a realization 'wind high, sun low'). Also it should easily be possible to vary the demand series as well as it is usually uncertain (e.g. realization 'Volcanic Winter' which implies higher heating costs).
- Enable plotting and reporting for decomposition methods: Currently the functions *Plotting* and *Reporting* are only defined for the original problem (without decomposition). It would be good to include that functionality for each decomposition method as well.
  - For Regional it should be straight forward to reuse the functionality from the original to do plotting and reporting for the sub regions. The master problem then just needs to add up the values from the sub problems.
  - For Divide Timesteps it would be necessary to patch the sub problems together to obtain a plot for the overall problem.
  - For SDDP plotting is more complex, because additionally to patching the sub problems together one needs to think about which path of realizations (or even some combination of paths) to plot.

Using the code for different models

There are several ways in which one could insert its own model or use parts of the code.

Create your own input file and use it with the urbs model

This is straightforward. Just take one of the existing input files and modify it by adding or removing sites, processes, commodities, transmissions, storage, demands, or supim commodities and setting their values. You need to make sure that you don't remove or add whole features, because the features and corresponding constraints are hardcoded in the models, so that would need to be changed (see further down).

Creating realizations for SDDP

If you want to create your own realizations which add uncertainty you need to modify the files `runme.py` and `sddp_sub`. When setting the SDDP parameters you can set a dict of probabilities and a list of factors for each scenario. The probabilities determine how likely a scenario is while the factor determines the changes to the supim series. In `create_uncertainty_data()` in `sddp_sub` this factor is used to modify the wind supim series. To create your own scenarios you can easily change the supim series to be modified or add several series by passing several numbers in the factor. This requires only small changes in the code of `create_uncertainty_data()`.

Use one of the decomposition methods but not for the urbs model

This would require major restructuring of the code, because it can only be used partly. All the parts which can be used are in the file `runme.py`. You can reuse the structure of this file for your own model. Basically you can use the main function and the function `run_scenario_decomposition` by making only relatively small changes. You'd need to modify all parts (or possibly just removing some) which are connected to input and output (reading data, loading and saving models, plotting and reporting) and create own master and sub instances instead of the urbs instances. For this you'd need do provide your own master and sub models for the corresponding decomposition method. The models must provide methods for cut generation and boundary setting. For an idea on how to split the model variables you can orient yourself at the existing decomposition methods:

- Divide Timesteps: In this method the master contains all variables which are independent of time and the subs all other variables.
- Regional decomposition: In this method the master contains all transmission variables while each sub problem contains all variables in its region.
- SDDP: This method works like Divide Timesteps but additionally introduces uncertainty on the supim time series.

If you are reusing one of the existing method it is possible to reuse the benders loop of the corresponding method. You then need to adjust the setting of boundaries and the upper bound calculation for your own variables. Also you can use the existing cut generation and adjust it for your own model.

Extend or delete a feature from the urbs model

If you want to extend the model by a feature you have to be careful to include it in all relevant parts of the code. Likewise you have to delete it in all relevant parts in case of deletion. Because of the similarity, only extension is explained. The relevant parts are:

- The input file
- In the model:
  - In the model preparation, at the start of `create_model()` in `super.py`.
  - In the model itself: For each decomposition method you need to choose whether the feature is included in the master model, the sub models or both. You then need to add the feature in the appropriate location (see *Extending the model structure*).
  - In the cut generation: If the feature includes a constraint in the sub models, the dual of the constraint needs to be taken into account for the cut generation.
- In `runme.py`
  - In boundary settings: If the feature include a variable in the master problem that imposes a restriction on the sub problems.
  - In upper bound calculations: If the feature introduces a new constraint or costs that are relevant for the upper bound calculation.

Extending the model structure

This section explains, in which class to put changes to the model structure.

- If adding something which is the same for all decomposition methods and the same for Master and Sub: add in super.py.
- If adding something specific to the Normal model: in `normal.py`.
- If adding something specific to a certain decomposition method and equivalent for Master and Subs: in `divide_timesteps_super.py` or `regional_super.py` or `sddp_super.py`.
- If adding something specific to the Master instance of a certain decomposition method: in `divide_timesteps_master.py` or `regional_master.py` or `sddp_master.py`.
- If adding something specific to the Sub instances of a certain decomposition method: in `divide_timesteps_sub.py` or `regional_sub.py` or `sddp_sub.py`.

Although this seems pretty simple, the disadvantage is when adding something which is e.g. the same for all master instances it has to be added in all 3 classes. This could be avoided by adding an additional class which summarizes all master classes, but then likewise a class would be necessary that summarizes all sub classes, then perhaps one that summarizes the subs and the normal and so on. This would become quite confusing. For this reason the classes were chosen like this, because it allowed for a maximum reduction in code duplicates (at least for the models at the time of creation) while keeping the class structure reasonably simple.

Perhaps it would be possible to further reduce duplicates while keeping the structure simple by creating a block structure, where features are encapsulated in small blocks of code that can then be added to the models as needed. In this case there would be no super classes, but a file which contains all these blocks. This though would be a big change to the code and probably be challenging.

Creating a new decomposition method

The current structure is somewhat ill suited to include a new decomposition method. It would be desirable to make the new decomposition method have the same structure as the other methods, that is, a master and a sub class which inherit from a super class which itself inherits from `ModelSuper`. The problem is that this would make restructuring of the code necessary in the following way: If there is a feature in the new decomposition method which is not included in both master and sub class but is included in `ModelSuper`, this feature would need to be removed from `ModelSuper`. Because the other decomposition methods still need to use that feature it would need to be passed down to all other classes which are next in the model hierarchy (e.g. to `DivideTimestepsSuper` and to `RegionalSuper` and to `SddpSuper` and to `Normal`).

Unification with urbs master branch

Differences to the urbs master branch

Compared to the urbs master branch there are some features missing in decomposition:

- Buy/Sell
- Demand Side Management

- Startup

One other big difference is the modularization of parts of the model:

Ideas how to combine decomposition models with modular urbs

The urbs master branch is using modular features that means the features are added in separate files which are called while creating the model. The big challenge to use this modularization for the decomposition branch as well will be that some features will look slightly different depending on the decomposition method and whether the model is a sub, a master or a normal problem.

As an example consider the feature `transmission.py`. In Regional the sub problem will not have the transmission capacity variables while the master problem will have them.

To resolve this it would be necessary to distinguish between different decomposition methods and model types within the features. This approach would be straight forward but quite cumbersome. Perhaps a more elegant approach would be to have a rule that could prohibit the use of certain variables within the feature. Then the feature could be called from any model by passing a list of the prohibited variables. This for example can already be realized for the capacity constraints (see *Sets, Variables, Parameters and Expressions* last bullet point) by setting the expressions and relax parameters correctly. Maybe this can be done for other constraints as well.

CHAPTER 2

Features

- urbs is a linear programming model for multi-commodity energy systems with a focus on optimal storage sizing and use.
- It finds the minimum cost energy system to satisfy given demand timeseries for possibly multiple commodities (e.g. electricity).
- By default, operates on hourly-spaced timesteps (configurable).
- 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 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

3.2 2016-08-18 Version 0.6

- *Demand Side Management Constraints* added
- *Partial & Startup Process Constraints* added
- Various fixes in examples, docs and tutorials for Pyomo 4/Python 3 changes

3.3 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: *Buy-Sell Documentation* long explanation for *Buy* and *Sell* commodity types
- Documentation: *Mathematical Documentation* full listing of sets, parameter, variables, objective function and constraints in mathematical notation and textual expanation

- 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

3.4 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"

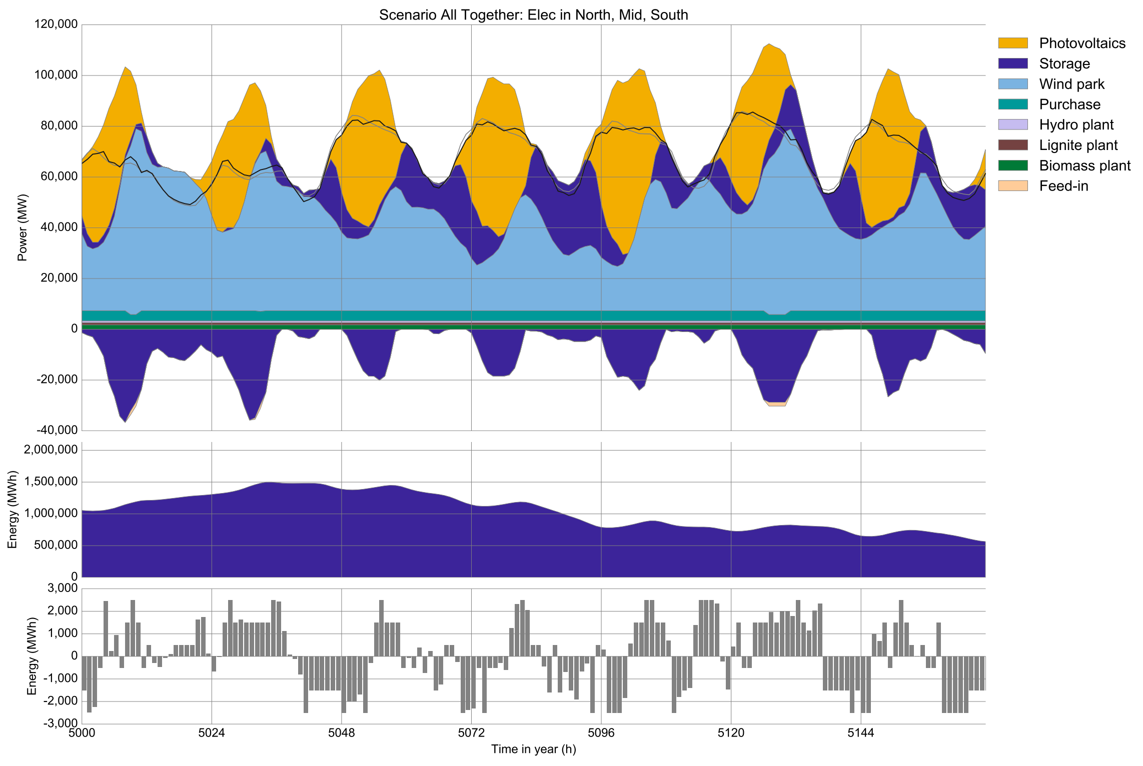
3.5 2014-12-05 Version 0.3

- 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`.

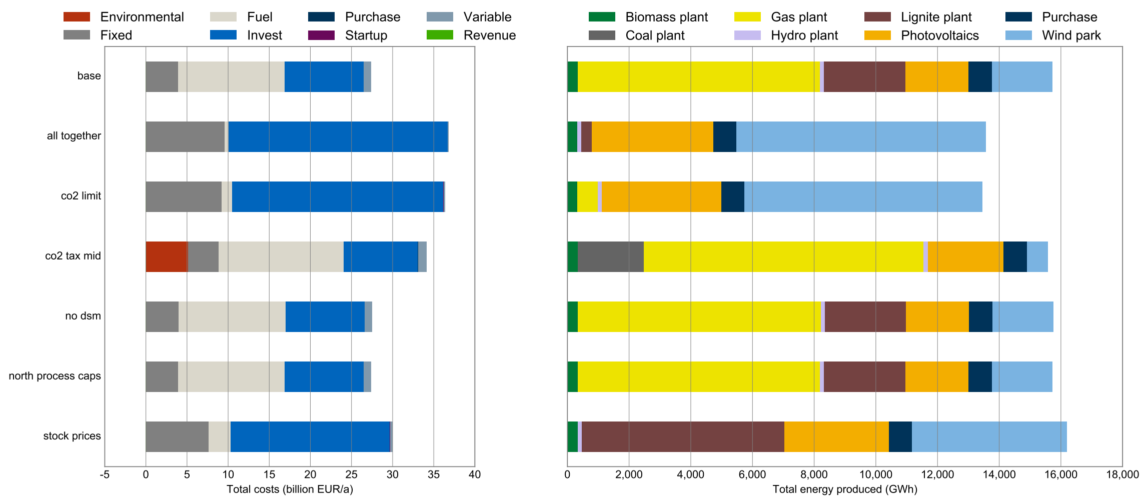
CHAPTER 4

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

Index