How to ZEN-garden

Table of Contents

1. Setup

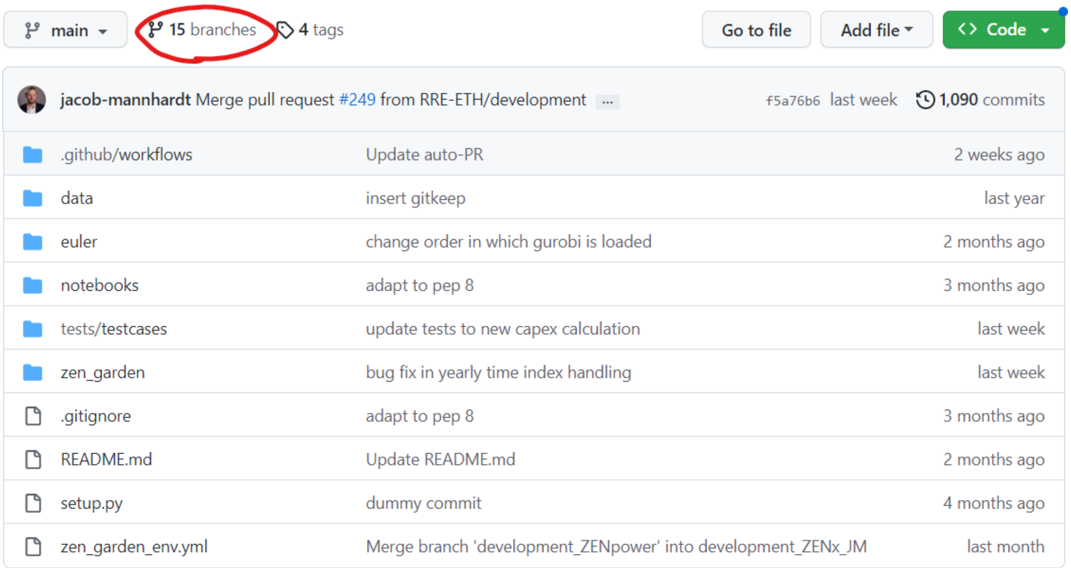
1.1. Needed Installations

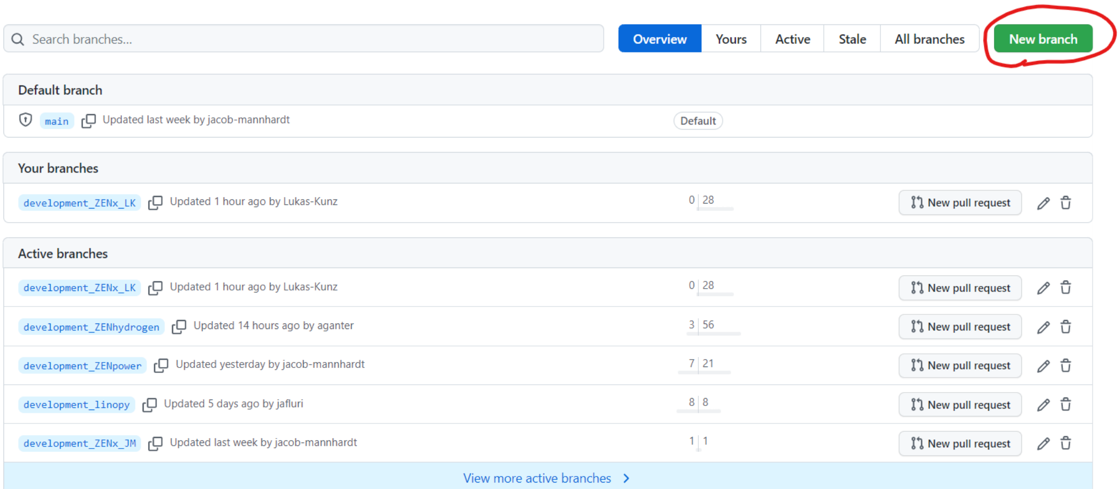
- PyCharm (IDE, you can use other IDEs as well, but most users of ZEN-garden use PyCharm) Install PyCharm
- Anaconda (Needed for ZEN-garden environment creation) Install Anaconda
- Gurobi (Optimization Software) Install Gurobi
- (GitHub Desktop) Install GitHub Desktop

1.2. Steps

1. GitHub registration: If you don't have a GitHub account yet register at: GitHub

2. Join ZEN-garden repository: If you didn't receive a GitHub invitation, ask your supervisor to invite you to the repository (write them your GitHub email address) ZEN-Garden Repository
3. Create your own branch: In the ZEN-garden repository click on "branches" and then "new branch", choose "main" as the branch source and "development_ZENx_NS" (NS= name, surname) as its name





4. Cloning the repository: To create a local copy of your branch on your computer, you must clone the remote repository from GitHub. It is important that you clone the repository to a path which doesn't contain any spaces! (Don't clone to e.g. ./Users/Name Surname, otherwise you'll have issues while executing the framework)To clone your branch there's a more beginner-friendly way using GitHub Desktop and a more advanced way using Git Bash for example.
   GitHub Desktop: Clone Reposiotry with GitHub Desktop

   To clone the repositry by using Git Bash, two methods are available: HTTPS or SSH

5. ZEN-garden environment creation: Open PyCharm to view the *zen_garden_env.yml* file (contained in the ZEN-garden folder), copy "*conda env create -f zen_garden_env.yml"* and run the command in your Anaconda prompt (takes several minutes), if the installation was successful, you can see the environment at *C:\Users\username\anaconda3\envs* or wherever Anaconda is installed
6. Gurobi license: To use all of Gurobi's functionalities, you need to obtain a free academic license: Get your Gurobi license
   Following these instructions, you'll get a Gurobi license key which you have to run in your command prompt to activate the license for your computer
7. Create PyCharm Configurations: To execute ZEN-garden's different functionalities configurations are used. To add them, follow the steps at "PyCharm Setup": Create Configurations

2. ZEN-garden configurations

2.1. Run ZEN-garden module

The ZEN-garden module can be executed in several ways as well as on ETH's EULER cluster. To check if the setup was successful, you can run one of the standardized test cases. To do so,

- open *ZEN-garden\tests\testcases* and copy paste all the tests to *ZEN-garden\data*
- copy *config.py* to the *data* directory
- choose "test_1a" as the dataset and execute it using one of the following methods

2.1.1. Run ZEN-garden using the "Run Module" configuration

Executing ZEN-garden with the created configuration "Run Module" (created in setup step 7) is the most forward way if you use PyCharm. Simply adjust the path in the analysis attribute "dataset" in the config.py file to one of the desired datasets and click the green run-button (have a look at the config options to get an overview of all the config settings).

```python
from zen_garden.model import Config
import os

# create a config
config = Config()

# Analysis - Default dictionary
analysis = config.analysis
# Solver - Default dictionary
solver = config.solver
# Scenarios - Default scenario dictionary
scenarios = config.scenarios

# Analysis - settings update compared to default values
analysis["dataset"]                          = os.path.join(os.path.dirname(__file__),
                                                            "test_1a")
analysis["objective"]                        = "total_cost"                # choose from "TotalCost", "TotalCarbonEmissions", "Risk"
analysis["use_existing_capacities"]            = True                      # use greenfield or brownfield approach
analysis["output_format"] = "h5"
# Solver - settings update compared to default values
#
solver["name"] = "gurobi_persistent" # free solver
# solver["solver_options"]["method"]       = 2
# solver["solver_options"]["node_method"]    = 2
solver["solver_options"]["bar_homogeneous"]    = 1
# solver["solver_options"]["presolve"]      = -1
solver["solver_options"]["threads"]       = 46
# solver["solver_options"]["crossover_basis"]    = 0
# solver["solver_options"]["crossover"]     = 0
solver["solver_options"]["scale_flag"]     = 2
solver["analyze_numerics"]                = True
solver["immutable_unit"]                  = ["hour","km"]
```

2.1.2. Run ZEN-garden using a terminal

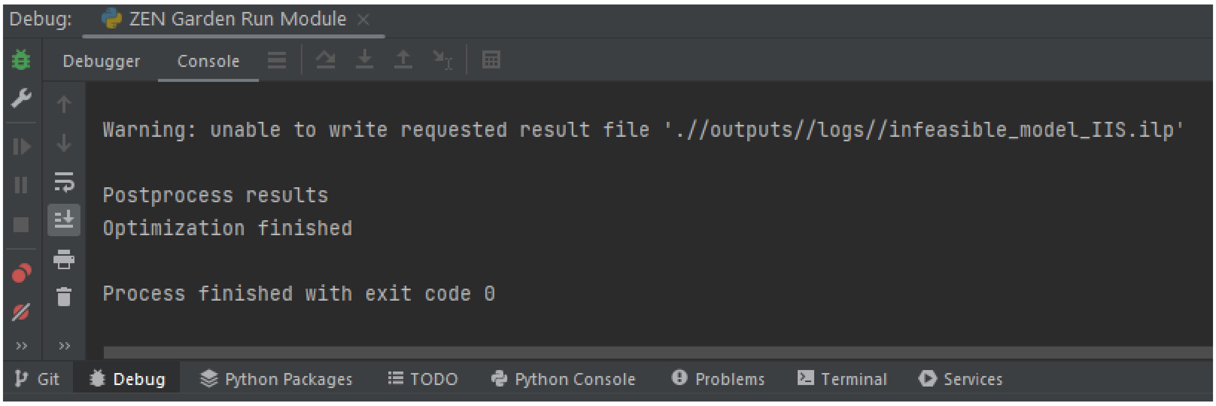
How to run the ZEN-garden package in a terminal is described in ZEN-garden as a package. Depending on the terminal you want to use, the procedure differs slightly. Before entering the module's execution command, ensure that the *data* folder is your working directory. To change the working directory from, e.g., *ZEN-garden* to *ZEN-garden/data*, simply run *cd data*.

PyCharm Power Shell (Terminal in PyCharm): As the zen-garden conda environment is activated by default, you can simply enter the following command followed by a chosen dataset name:

(zen-garden) PS C:\Users\Lukas Kunz\ETH\ZEN_garden\ZEN-garden\data> python -m zen_garden --dataset="test_1b"

Anaconda Prompt: The only difference when using the Anaconda prompt is that you have to activate the zen-garden environment manually before you can run the package execution command. This can be done by running *conda activate zen-garden.*

If your console looks something like the screenshot below, the ZEN-garden module works fine on your computer, and you can run all the data sets located in the *data* folder by choosing one of the two methods. Otherwise, revisit the setup steps according to the occurred error.



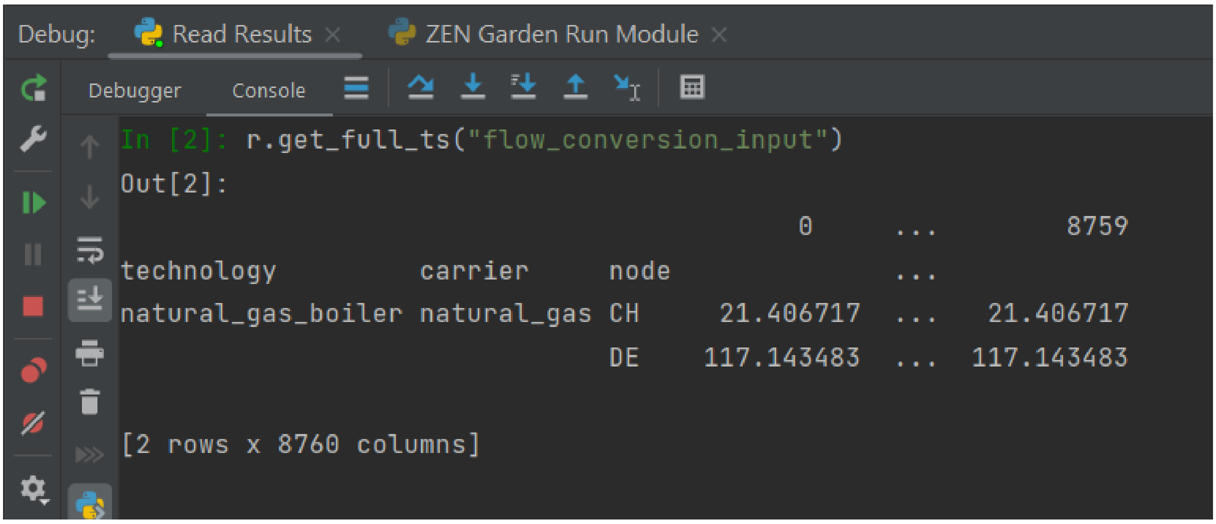
2.1.3. Run ZEN-garden on EULER

To run computational more expensive optimization problems, ETH's EULER cluster can be accessed as described at [ZEN-garden on EULER].

2.2. Read Results

After a dataset's optimization problem has been executed, its results can be accessed and visualized with help of the *results.py* script. To get a first impression of the available results processing functionalities, the Jupyter Notebook *postprocess_results.ipynb* can help a lot. It can be found in ZEN-garden's *notebooks* directory.

Another way to access your results is to use the "Read Results" configuration. By running the *results.py* script, the different member functions of the contained "Results" class can be applied to the "Results" object to extract and plot the data of your optimization problem. Since the "Read Results" configuration creates an instance of the "Results" class, the object can be accessed by "self". By setting a break point at the end of the file, the debugger console can be used to apply the class's functions to the "Results" instance.



2.2.1. How to plot your results

The class "Results" contains three member functions to plot the simulated data. Please have a look at [the plot discussion entry] to get further explanations.

2.2.2. Accessing your results data

To access the data frames containing the raw optimization results of the variables and parameters, the following member functions of the "Results" class can be used:

1. r.get_total(),
2. r.get_full_ts(),
3. r.get_df(),

where "r" is an instance of the "Results" class. r.get_total() returns the aggregated values of the variable and parameter values for each year. For hourly resolved variables, such as "flow_conversion_input", this is the sum over all hours of the year for each year. Yearly resolved variables, such as "capacity", remain unaltered because they are already yearly aggregates.

r.get_full_ts() returns the hourly evolution of hourly resolved variables. This is especially useful when using the time series aggregation, where the hours of the year are aggregated by representative time steps. r.get_full_ts() disaggregates the time series back to full hourly representation. Yearly values remain in yearly resolution, thus for these components r.get_total() and r.get_full_ts() return the same result.

Under the hood of r.get_total() and r.get_full_ts(), we use the r.get_df() function to extract the raw variable and parameter values. If these are of interest, you can use r.get_df(), otherwise r.get_total() and r.get_full_ts() will be more useful.

2.2.2.1. self.get_df()

The most fundamental function to access the data of a specific variable such as, e.g., “flow_conversion_input” is *self.get_df(“flow_conversion_input”)*. It returns a Pandas series containing all the “flow_conversion_input” values of the different technologies at the individual nodes at every time step.

hard_coal_plant/hard_coal/FR/144	0.00000
hard_coal_plant/hard_coal/FR/145	0.00000
hard_coal_plant/hard_coal/FR/146	0.00000
hard_coal_plant/hard_coal/FR/147	0.00000
hard_coal_plant/hard_coal/FR/148	0.00000
hard_coal_plant/hard_coal/FR/149	0.00000
nuclear/uranium/CH/0	12.54420
nuclear/uranium/CH/1	12.54420
nuclear/uranium/CH/2	8.05693
nuclear/uranium/CH/3	12.54420
nuclear/uranium/CH/4	12.54420
nuclear/uranium/CH/5	12.54420
nuclear/uranium/CH/6	12.54420
nuclear/uranium/CH/7	12.54420

2.2.2.2. self.get_full_ts()

A more convenient way to access the same data is offered by *self.get_full_ts(“flow_conversion_input”)*, a function which creates a data frame of the variable’s full time series.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
biomass_plant/biomass/FR	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632	3.03632
electrode_boiler/electricity/CH	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
electrode_boiler/electricity/DE	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
electrode_boiler/electricity/FR	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
hard_coal_boiler/hard_coal/CH	0.15540	0.15540	0.15540	0.18068	0.18068	0.22886	0.22886	0.22886	0.18068	0.18068	0.18068	0.18068	0.18068	0.18068	0.18068	0.18068
hard_coal_boiler/hard_coal/DE	3.28069	3.28069	3.28069	0.00000	0.00000	4.78539	4.78539	4.78539	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
hard_coal_boiler/hard_coal/FR	0.00000	0.00000	0.00000	0.00000	0.00000	0.38259	0.38259	0.38259	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
hard_coal_plant/hard_coal/CH	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
hard_coal_plant/hard_coal/DE	0.00000	0.00000	0.00000	17.43686	17.43686	12.65147	12.65147	12.65147	17.43686	17.43686	17.43686	17.43686	17.43686	17.43686	17.43686	17.43686
hard_coal_plant/hard_coal/FR	1.68525	1.68525	1.68525	1.68525	1.68525	1.30265	1.30265	1.30265	1.68525	1.68525	1.68525	1.68525	1.68525	1.68525	1.68525	1.68525
heat_pump/electricity/CH	2.91542	2.91542	2.91542	3.58026	3.58026	4.84721	4.84721	4.84721	3.58026	3.58026	3.58026	3.58026	3.58026	3.58026	3.58026	3.58026
heat_pump/electricity/DE	18.19193	18.19193	18.19193	23.78045	23.78045	31.87778	31.87778	31.87778	23.78045	23.78045	23.78045	23.78045	23.78045	23.78045	23.78045	23.78045
heat_pump/electricity/FR	13.92653	13.92653	13.92653	17.69159	17.69159	24.85001	24.85001	24.85001	17.69159	17.69159	17.69159	17.69159	17.69159	17.69159	17.69159	17.69159
lignite_coal_plant/lignite/CH	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
lignite_coal_plant/lignite/DE	0.00000	0.00000	0.00000	30.95859	30.95859	30.95859	30.95859	30.95859	30.95859	30.95859	30.95859	30.95859	30.95859	30.95859	30.95859	30.95859
lignite_coal_plant/lignite/FR	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
atural_gas_boiler/natural_gas/CH	0.00000	0.00000	0.00000	0.00000	0.00000	2.29462	2.29462	2.29462	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
atural_gas_boiler/natural_gas/DE	0.00000	0.00000	0.00000	0.00000	0.00000	1.83633	1.83633	1.83633	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
atural_gas_boiler/natural_gas/FR	0.00000	0.00000	0.00000	0.00000	0.00000	1.91709	1.91709	1.91709	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
tural_gas_turbine/natural_gas/CH	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
tural_gas_turbine/natural_gas/DE	0.00000	0.00000	0.00000	0.00000	0.00000	18.10258	18.10258	18.10258	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
tural_gas_turbine/natural_gas/FR	0.00000	0.00000	0.00000	0.94681	0.94681	7.20783	7.20783	7.20783	0.94681	0.94681	0.94681	0.94681	0.94681	0.94681	0.94681	0.94681
nuclear/uranium/CH	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420	12.54420
nuclear/uranium/DE	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843	76.19843
nuclear/uranium/FR	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000	171.81000

2.2.2.3. self.get_total()

If you're not interested in the hourly resolution of the variable values, *self.get_total("flow_conversion_input")* can be used to obtain the yearly sums of the hourly data.

	0	1	2
biomass_plant/biomass/FR	6928.874...	6385.374...	6278.759...
electrode_boiler/electricity/CH	500.99039	1438.128...	1497.060...
electrode_boiler/electricity/DE	6826.321...	5974.843...	5830.382...
electrode_boiler/electricity/FR	19313.41...	31253.19...	34107.87...
hard_coal_boiler/hard_coal/CH	703.04750	696.99206	696.99206
hard_coal_boiler/hard_coal/DE	19835.34...	19741.43...	19520.49...
hard_coal_boiler/hard_coal/FR	783.73352	783.73352	1002.791...
hard_coal_plant/hard_coal/CH	0.00000	0.00000	0.00000
hard_coal_plant/hard_coal/DE	40596.64...	40199.47...	40257.89...
hard_coal_plant/hard_coal/FR	2585.724...	2726.462...	2478.375...
heat_pump/electricity/CH	18526.75...	18755.79...	18755.79...
heat_pump/electricity/DE	128462.6...	129016.8...	129016.8...
heat_pump/electricity/FR	98406.84...	98406.84...	98522.42...
lignite_coal_plant/lignite/CH	0.00000	0.00000	0.00000
lignite_coal_plant/lignite/DE	117739.8...	115236.4...	113795.9...
lignite_coal_plant/lignite/FR	0.00000	0.00000	0.00000
natural_gas_boiler/natural_gas/CH	2353.467...	2166.532...	2166.532...
natural_gas_boiler/natural_gas/DE	3429.868...	3003.047...	3324.969...
natural_gas_boiler/natural_gas/FR	1574.848...	1728.270...	1624.832...
natural_gas_turbine/natural_gas/CH	0.00000	0.00000	0.00000
natural_gas_turbine/natural_gas/DE	34633.01...	34296.42...	33245.51...
natural_gas_turbine/natural_gas/FR	4576.558...	5035.394...	5534.793...
nuclear/uranium/CH	92638.35...	93954.40...	87898.20...
nuclear/uranium/DE	645155.8...	696369.0...	723363.1...
nuclear/uranium/FR	1273541....	1321776....	1327109....

2.2.3. Compare two datasets

You can compare two "Results" objects by using the following class methods. They can help you to get a fast overview of two datasets' differences which facilitates spotting the reasons for errors. Again, the Jupyter Notebook shows some practical examples, but the functionalities can be used in the debug console as well by creating the desired "Results" objects the way it is done in the beginning of the notebook. All the functions take a list of two "Results" instances as their input argument.

- relate the configs of two datasets:
  Results.compare_configs([result_instance_1, result_instance_2])
- relate model parameters:
  Results.compare_model_parameters([result_instance_1, result_instance_2])
- relate model variables:
  Results.compare_model_variables([result_instance_1, result_instance_2])

2.3. Run Tests

The main purpose of the test files is their usage for the automated testing functionality of ZEN-garden. By comparing the variables' values gathered by simulating the testcases with some reference values, the correctness of the current framework code can be proved. Whenever you adapted some framework code, you can use the run test configuration to ensure that ZEN-garden does still function properly.

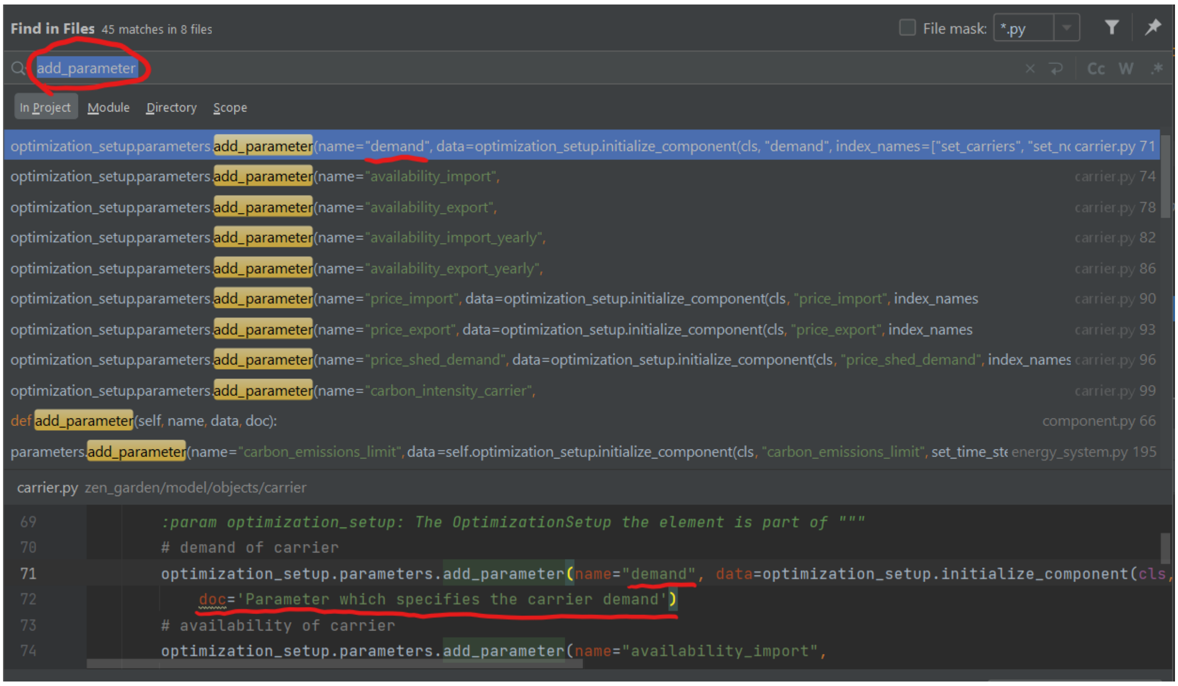
3. Parameters, variables, and constraints

An important concept in ZEN-garden, or for optimization problems in general, is the definition of parameters, variables, and constraints. Parameters are used to store data that is immutable, meaning once a parameter's values are specified, they stay the same for the whole optimization (e.g., the hourly electricity demand per country). On the other hand, variables represent quantities whose values are computed by solving the optimization problem (e.g., the hourly electricity output flow of a gas turbine). By defining constraints, the parameters and variables can be related to each other such that they follow the rules of physical properties etc. (e.g., energy conservation at nodes). In the example optimization problem below, $c^T x$ is the so-called objective function whose value is optimized (mostly minimizing the net present cost of the entire system), x and b are vectors containing all the variables and parameters, respectively, which are related by constraints of the form $Ax \leq b$. Additionally, some variables are defined as non-negative numbers, i.e., $x \geq 0$, as physical metrics like costs, power flows and energy etc. can only be positive.

$$\begin{aligned}&\min_x c^T x\\ &s.t. Ax \leq b\\ &\qquad x \geq 0\end{aligned}$$

To get an overview of all the existing parameters and variables you can access your "Results" object in your debugger or in the Jupyter Notebook, which contains the dictionary *component_names*, which itself contains the dictionaries *pars* and *vars*. Inside these dictionaries, you can find all the parameters'/variables' names.

To find the definitions of all the parameters, variables and constraints you can look up every appearance of *add_parameter/add_variable/add_constraint* in all of ZEN-garden's files by using CTRL+Shift+F. Assessing the definitions can be quite helpful to get a better understanding as they include the *doc* strings, a brief explanation of the underlying parameter, variable or constraint. In addition, it can be seen in which file *(technology.py*, *carrier.py*, etc.) the definition is located, revealing some extra information. Since this method takes some time to find the desired doc string, the *Results* class contains the function *r.get_doc("component")* which returns the doc string of the corresponding component.

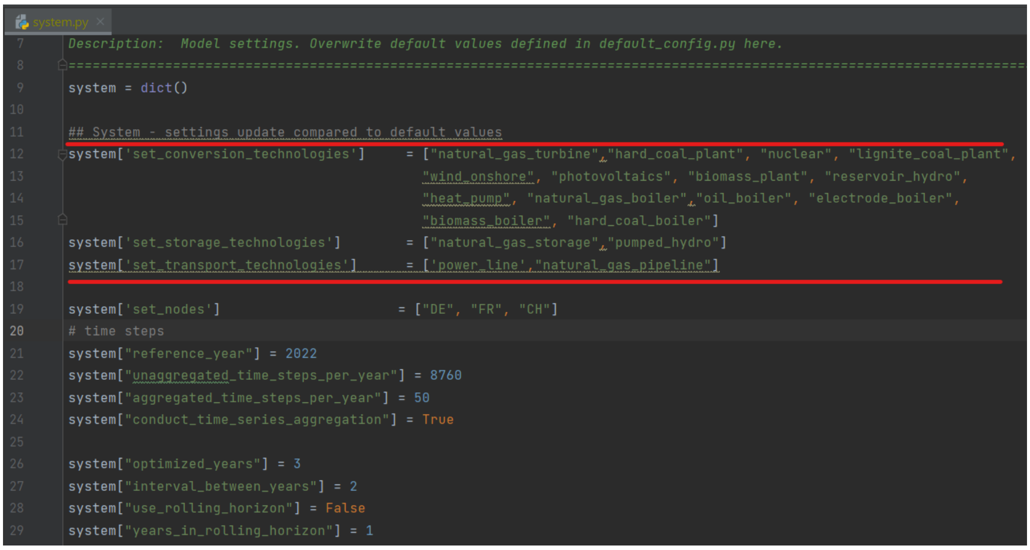


4. Input data structure

The input data of a dataset must be composed of the *system.py* file and the five folders *set_carriers*, *set_conversion_technologies*, *set_transport_technologies*, *set_storage_technologies* and *system_specification*.

4.1. system.py

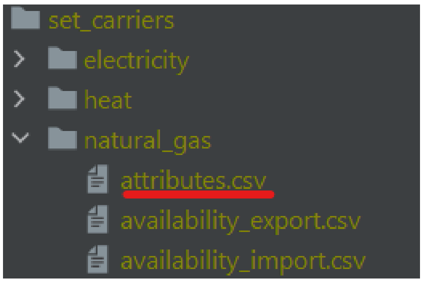
The *system.py* file must contain the sets of technologies that constitute the energy system, i.e. , that take part in supplying the final energy demands. You can have technologies in your input data folder but not list them in the system.py. In this case, they are excluded from the optimization. Additionally, a subset of nodes (from *system_specification/set_nodes.csv*), the starting year of the optimization (*reference_year*) and a lot of other time related specifications can be defined. The time step parameters are discussed in this Git Discussion. To get an overview of how to define the different properties, have a look at the system settings.



4.2. set_carriers

The *set_carriers* folder contains the energy carrier types such as electricity, heat, biomass, natural gas, etc. All the carriers that are needed by the technologies specified in the *system.py* file must be contained in this directory; additional carriers are allowed as well. You do not need to specifically list the carriers in system.py as they are implied by the included technologies.

To define a specific carrier, a folder named after the carrier containing the attributes file must be created.



The attributes file contains all the default values of the parameters' needed to describe the carrier. As the parameters' values can differ along the energy systems' nodes and the simulated time steps, the variations can be described by creating additional input files having the following name structure (parameter name without the "default" ending):

- **demand.csv:** If there exists a demand for a carrier, it can be described in the demand file.
- **availability_import.csv:** This file can be used to specify different values of a carrier's import availability as it may differ for the nodes, time steps etc.
- **availability_export.csv:** As for the import availability the export availability can be customised.
- **…**

Examples of existing parameters can be assessed in the attribute files of the test datasets (for completion, the whole set of parameters can be found in the appendices). To get a better understanding of how to structure these additional input files, have a look at the "Spreadsheet structure" section.

attributes.csv

```
index,value,unit
carbon_intensity_default,0.0,kilotons/GWh
demand_default,0.0,GW
price_shed_demand_default,inf,kiloEuro/GWh
availability_import_default,0.0,GW
availability_export_default,0.0,GW
availability_import_yearly_default,inf,GWh
availability_export_yearly_default,inf,GWh
price_export_default,0.0,kiloEuro/GWh
price_import_default,0.0,kiloEuro/GWh
```

4.3. set_conversion_technologies

The *set_conversion_technologies* folder contains the energy conversion technologies such as boilers, power plants (e.g., lignite coal plants), or renewables. All the conversion technologies that are specified in the system file's technology sets must be contained in this directory; additional conversion technologies are allowed. The procedure of defining a specific conversion technology is the very same as for energy carriers, described in the previous section. Again, a folder with the conversion technology's name must be created, including the attributes file for conversion technologies and variations in space and time can be specified with additional input data files.

attributes.csv

```
index,value,unit
capacity_addition_min_default,0,GW
capacity_addition_max_default,inf,GW
capacity_existing_default,0,GW
capacity_limit_default,inf,GW
min_load_default,0,
max_load_default,1,
lifetime_default,25,
opex_specific_variable_default,0,kiloEuro/GWh
reference_carrier,heat,GW
carbon_intensity_default,0,kilotons/GWh
construction_time_default,2,
capacity_investment_existing_default,0,GW
opex_specific_fixed_default,87.6,Euro/kW
max_diffusion_rate_default,inf,
input_carrier,natural_gas,GW
output_carrier,heat,GW
conversion_factor_default,1.1,
capex_specific_default,876,Euro/kW
capacity_addition_unbounded_default,0,GW
```

4.4. set_transport_technologies

The *set_transport_technologies* folder contains the energy transport technologies such as natural gas pipelines or power lines. All the transport technologies that are specified in the system file's technology sets must be contained in this directory; additional transport technologies are allowed. Once more, the individual transport technologies must be defined the same way as carriers and other technologies, however, they additionally need the *distance.csv* file. This file is needed to define the distances between the node pairs (e.g., needed to compute resistive losses of power lines).

distance.csv

```
edge,distance
AT-CH,453.6972189390188
AT-CZ,255.4670495934849
AT-DE,474.5969264386361
AT-HU,402.2556618433108
AT-IT,543.0103186128234
AT-SI,170.35958789572155
AT-SK,417.69792800060645
```

4.5. set_storage_technologies

The *set_storage_technologies* folder contains the energy storage technologies such as pumped hydro, natural gas storages, batteries, etc. All the storage technologies that are specified in the system file's technology sets must be contained in this directory; additional storage technologies are allowed. Again, the procedure of defining them is equivalent as before.

4.6. system_specification

The *system_specification* folder contains additional input data that is needed to define the energy system as a whole. Other than the carrier and technology folders, this folder must contain more files than just the attributes file:

- attributes.csv: carbon emissions related information etc.
- base_units.csv: definition of base units to which input data units are converted
- set_edges.csv: definition of existing connections (edges) between node pairs
- set_nodes.csv: set of all nodes (used when system file doesn't contain a subset of nodes)
- unit_definitions.txt: definition of additional units

4.7. Spreadsheet structure

The individual values at different nodes and time steps can be entered into the input files by using the column headers "node" and "time"/"year" as it is done in the pictures.

availability_import.csv

```
node,availability_import
AT,3.828615867579908
BE,3.9944747885844747
BG,4.347786643835617
CH,3.828615867579908
CY,0.17639429223744293
CZ,5.55635194063927
DE,23.043983105022832
```

price_import_yearly_variation.csv

```
year,price_import
2022,1.0
2026,1.3538461538461541
2030,2.7692307692307696
2034,2.2256410256410257
2038,1.6820512820512823
2042,1.4102564102564104
2046,1.4102564102564104
2050,1.4102564102564104
```

The header "time" is used to represent hourly time steps, whereas the "year" header serves for yearly time steps (time step discussion). An overview of the parameters' time step types is given in the appendices. In addition to the one-dimensional input structure above, data varying in space (nodes) and time can be structured by stating the nodes and time steps explicitly:

demand_yearly_variation.csv

```
node,2022,2023,2024,2025,2026,2027,2028,2029,2030,2031,2032,2033,2034,2035,2036
AT,1.0,1.0325760178707748,1.0651520357415494,1.0977280536123244,1.130304071483099,1.1670
BE,1.0,1.0225383905733347,1.0450767811466697,1.067615171720004,1.090153562293339,1.11128
BG,1.0,1.0244532672401176,1.048906534480235,1.0733598017203527,1.0978130689604702,1.1162
CY,1.0,1.0410721932295168,1.0821443864590337,1.1232165796885505,1.164288772918067,1.1974
CZ,1.0,1.0271041191833636,1.0542082383667273,1.0813123575500911,1.1084164767334548,1.146
DE,1.0,1.030163215705968,1.0603264314119356,1.0904896471179033,1.120652862823871,1.14696
DK,1.0,1.0510992646188404,1.1021985292376808,1.153297793856521,1.2043970584753616,1.2415
EE,1.0,1.009735355797026,1.0194707115940518,1.0292060673910777,1.0389414231881031,1.0614
ES,1.0,1.0191958275721682,1.038391655144336,1.0575874827165042,1.076783310288672,1.10729
FI,1.0,1.0402840895544,1.0805681791088002,1.1208522686632003,1.1611363582176002,1.221652
FR,1.0,1.0214303876840425,1.0428607753680847,1.064291163052127,1.0857215507361695,1.1135
EL,1.0,1.01314191484352,1.0262838296870405,1.0394257445305606,1.052567659374081,1.084286
HR,1.0,1.0151770326027998,1.0303540652055996,1.0455310978083996,1.0607081304111994,1.073
HU,1.0,1.0293697138231863,1.058739427646373,1.0881091414695596,1.1174788552927462,1.1401
IE,1.0,1.0601821824730624,1.1203643649461248,1.180546547419187,1.2407287298922494,1.2736
```

Thanks to ZEN garden's capability of completing required parameter values which are not stated in the input files explicitly, the user doesn't have to specify the values for all indices of the parameter. For example, the parameter *availability_import* is defined by the *index_sets* "set_nodes" and "set_time_steps", however it is possible to only provide data for the individual nodes and none for the different time steps (see screen shot above). The input data handling will then complete the "missing" values by using the nodes' individual import availabilities for all the time steps (time independent parameter). Therefore, the framework user can choose for each parameter, if the default value (specified in the attributes file) or individual values for the parameter's index sets should be used.

carier.py

```
"] = self.data_input.extract_input_data("demand", index_sets=["set_nodes", "set_time_steps"], time_steps=set_base_time_
bility_import"] = self.data_input.extract_input_data("availability_import", index_sets=["set_nodes", "set_time_steps"],
bility_export"] = self.data_input.extract_input_data("availability_export", index_sets=["set_nodes", "set_time_steps"],
export"] = self.data_input.extract_input_data("price_export", index_sets=["set_nodes", "set_time_steps"], time_steps=se
import"] = self.data_input.extract_input_data("price_import", index_sets=["set_nodes", "set_time_steps"], time_steps=se
```

4.8. Additional methods to enter input data

4.8.1. PWA

To approximate nonlinear functions such as the conversion efficiency of e.g., heat pumps the so-called piecewise affine (PWA) approximation can be used. In this method, the nonlinear function is divided into several regions where it can be approximated as a linear function. To specify such PWA input data a similar combination of the following two files is needed (in the folder of the corresponding technology):

- *breakpoints_pwa_conversion_factor.csv* and *nonlinear_conversion_factor.csv* to approximate the nonlinear conversion efficiency of e.g., a natural gas boiler
- *breakpoints_pwa_coapex.csv* and *nonlinear_capex.csv* to approximate the nonlinear relation of capex and installed capacity of a technology
- …

The "nonlinear" file must thereby contain the values for the nonlinear relation between the two metrics (e.g., heat output vs. natural gas input) followed by a pair of units whereas the "breakpoints" file is used to divide the nonlinear function into several intervals (#intervals = #breakpoints - 1).

nonlinear_conversion_factor.csv

```
heat,natural_gas
0,0.0
70,77
140,150.5
GW,GW
```

breakpoints_pwa_conversion_factor.csv

```
heat
0
70
140
GW
```

4.8.2. Define technology with multiple input/output carriers

To define a conversion technology with multiple input and output carriers, several carrier types can be specified as the technology's input/output carriers in its corresponding attributes file (e.g., heat and carbon as output carrier). Having several input/output carriers requires additional conversion factors, i.e., the factor relating the amount generated/consumed of a carrier with respect to the one's of the reference carrier. Therefore, the *conversion_factor_default* needs to be overwritten by providing the additional *conversion_factor.csv* file, where all conversion factors need to be specified with respect to the *reference_carrier* (e.g., since the reference carrier in the screenshot is heat and the two other carriers are natural gas and carbon, the conversion factors relating heat with natural gas (1.1 GWh NG/GWh heat) and heat with carbon (0.01 kt carbon/GWh heat) need to be declared).

attributes.csv

```
index,value,unit
capacity_addition_min_default,0,GW
capacity_addition_max_default,inf,GW
capacity_existing_default,0,GW
capacity_limit_default,inf,GW
min_load_default,0,
max_load_default,1,
lifetime_default,25,
opex_specific_variable_default,0,kiloEuro/GWh
reference_carrier,heat,GW
carbon_intensity_default,0,kilotons/GWh
construction_time_default,0,
capacity_investment_existing_default,0,GW
opex_specific_fixed_default,87.6,Euro/kW
max_diffusion_rate_default,inf,
input_carrier,natural_gas,GW
output_carrier,heat carbon,GW
conversion_factor_default,1.1,
capex_specific_default,876,Euro/kW
capacity_addition_unbounded_default,0,GW
```

conversion_factor.csv

```
year,natural_gas,carbon
2022,1.1,0.01
unit,GWh/GWh,kilotons/GWh
```

4.8.3. Define input data with yearly variations

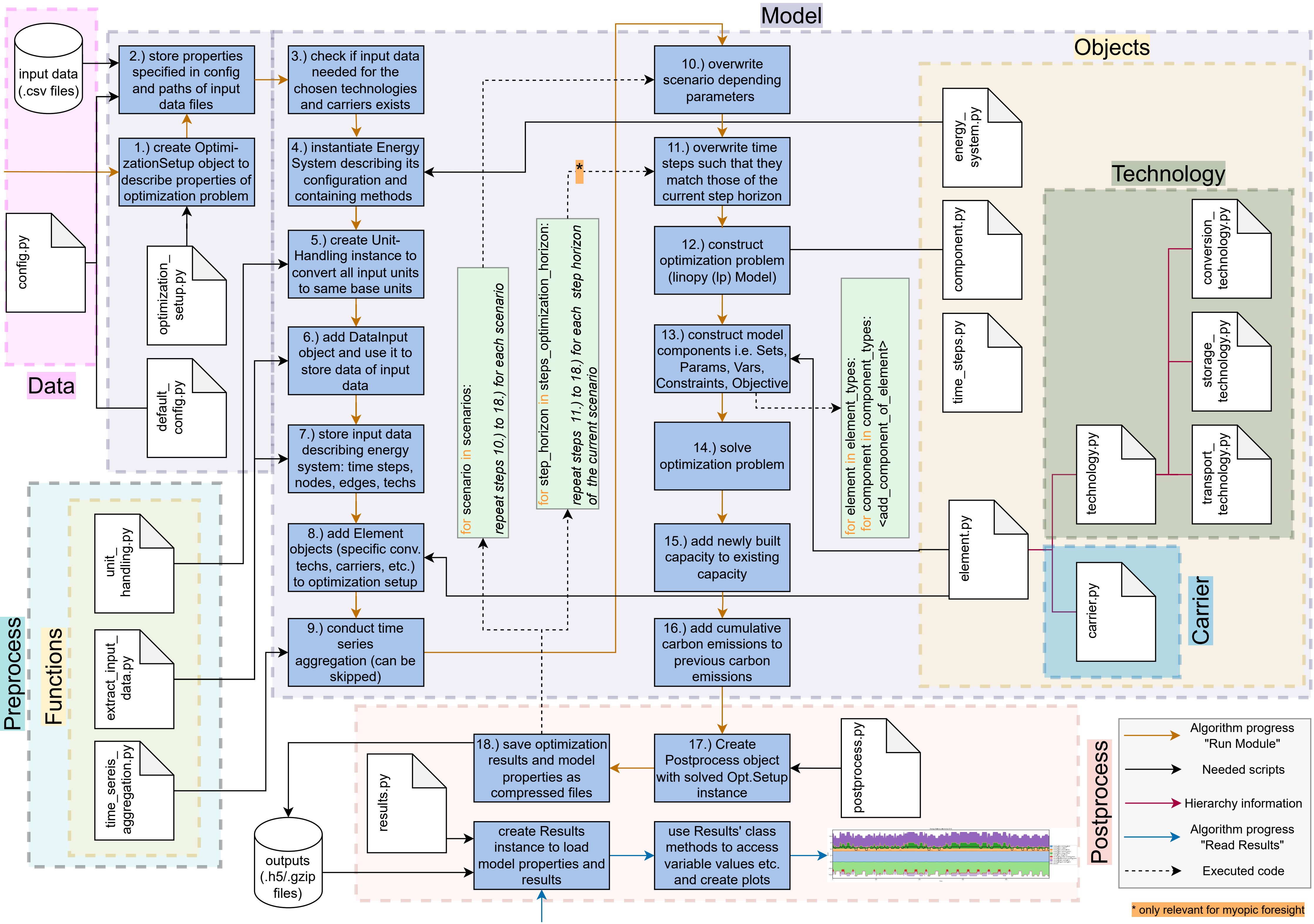
To simplify the hourly dependent input data which varies each year by a specific factor the “yearly_variation” file can be used. For example, if the heat demand is expected to increase or decrease over the years by a known percentage, the change can be specified as it is done in the following figure instead of defining all the values explicitly in the heat demand file. By doing so, the demand values will be scaled accordingly to the yearly variation factors (e.g., demand CH in year 2022 and time step 0 will be 9).

demand_yearly_variation_4.csv

```
year,CH,DE
2022,0.9,1.1
2023,1.0,1.2
```

demand.csv

```
time,CH,DE
0,10,100
1,20,90
```



Model
Objects
Technology
Carrier
Data
Preprocess
Functions
Postprocess
input data (.csv files)
config.py
optimization_ setup.py
default_ config.py
unit_ handling.py
extract_input_ data.py
time_sereis_ aggregation.py
1.) create OptimizationSetup object to describe properties of optimization problem
2.) store properties specified in config and paths of input data files
3.) check if input data needed for the chosen technologies and carriers exists
4.) instantiate Energy System describing its configuration and containing methods
5.) create Unit-Handling instance to convert all input units to same base units
6.) add DataInput object and use it to store data of input data
7.) store input data describing energy system: time steps, nodes, edges, techs
8.) add Element objects (specific conv. techs, carriers, etc.) to optimization setup
9.) conduct time series aggregation (can be skipped)
for scenario in scenarios: repeat steps 10.) to 18.) for each scenario
for step_horizon in steps_optimization_horizon: repeat steps 11.) to 18.) for each step horizon of the current scenario
*
10.) overwrite scenario depending parameters
11.) overwrite time steps such that they match those of the current step horizon
12.) construct optimization problem (linopy (lp) Model)
13.) construct model components i.e. Sets, Params, Vars, Constraints, Objective
14.) solve optimization problem
15.) add newly built capacity to existing capacity
16.) add cumulative carbon emissions to previous carbon emissions
17.) Create Postprocess object with solved Opt.Setup instance
18.) save optimization results and model properties as compressed files
for element in element_types: for component in component_types: <add_component_of_element>
energy_ system.py
component.py
time_steps.py
element.py
technology.py
conversion_ technology.py
storage_ technology.py
transport_ technology.py
carrier.py
postprocess.py
results.py
outputs (.h5/.gzip files)
create Results instance to load model properties and results
use Results' class methods to access variable values etc. and create plots
Algorithm progress "Run Module"
Needed scripts
Hierarchy information
Algorithm progress "Read Results"
Executed code
* only relevant for myopic foresight

5. Framework structure

ZEN-garden is structured into the three building blocks *preprocess*, *model* and *postprocess*. *model* is used to describe the optimization problem of the energy system containing the different technologies and carriers, *preprocess* extracts the provided input data and *postprocess* saves and visualizes the simulation results. To get a better understanding of the general order of ZEN-gardens execution steps and the package levels, have a look at the flowchart.

5.1. Preprocess

5.1.1. Functions

extract_input_data.py

The functions to extract the data from the differently structured spreadsheets and store the information in dataframes are in this script. As there are a lot of different ways in which the input data itself or the description of specific parameters can be specified (linear/PWA, from attributes/extra file, etc.), the data extraction process is quite complicated, leading to the large number of functions.

time_series_aggregation.py

To reduce the complexity and thus the computational cost of optimizing energy systems with hourly data resolution, the time series of the underlying data can be aggregated to decrease the number of time steps with individual data values. For example, *test_4b* aggregates the 8760 hourly time steps of a year to ten representative time steps, thus reducing the computational effort heavily while approximating the original data still well. The time series aggregation parameters are defined in the system file and further details can be found at: TSA discussion

unit_handling.py

To ensure that all the units of the input data are consistent, a unit-handling is implemented. By specifying a set of base units (*system_specification/base_units.csv*), the units used in the input data files do not have to be consistent as the unit handling will transform the units according to the chosen base units (unit handling discussion).

5.2. Model

default_config.py

The default configuration is defined in the *Config* class which describes all the relevant specifications for ZEN-garden with the four dictionaries:

- **analysis**: describes the desired analysis such as the objective (e.g., total cost), if the problem should be minimized or maximized, the discount rate, etc.
- **solver**: contains the information regarding the solver used to solve the optimization problem such as its name (e.g. glpk (free) or gurobi_persistent (commercial; free with academic license)) and different corresponding solver specifications (Gurobi documentation)
- **system**: describes the energy system; used to select technologies which are included in the optimization problem and to define a subset of nodes which should exist (more technologies and nodes allowed in input data; see above), contains time related parameters to specify time series aggregation (TSA discussion)
- **scenarios**: used to define the individual settings of multi-scenario simulations (see *test_6a*)

To specify changes from the default configuration the files *config.py*, *system.py* and *scenarios.py* are used.

- **config.py:** The config.py file is in the *data* and the *testcases* folder. The changes made in these files apply to all the datasets contained in the corresponding folder. For example, if you specify a solver name in the *data* config, it affects all the datasets located in the *data* directory. As described in the “Run ZEN-garden module” section, the config can be used to select which dataset should be executed.
- **system.py:** Parameters that describe the energy system are changed in the system.py file and only apply to the specific dataset in which the system.py file is located. Each dataset must contain its own system file which should look similar as:

```python
system = dict()

## System - settings update compared to default values
system['set_conversion_technologies']     = ["natural_gas_turbine","hard_coal_plant",
                                             "nuclear","lignite_coal_plant",
                                            "wind_onshore","photovoltaics",
                                            "biomass_plant","reservoir_hydro",
                                            "heat_pump","natural_gas_boiler","oil_boiler","electrode_boiler","biomass_boiler","hard_coal_boiler"]
system['set_storage_technologies']        = ["natural_gas_storage","pumped_hydro"]
system['set_transport_technologies']      = ['power_line',"natural_gas_pipeline"]

system['set_nodes']                      = ["DE","FR","CH"]
# time steps
system["reference_year"] = 2022
system["unaggregated_time_steps_per_year"] = 8760
system["aggregated_time_steps_per_year"] = 50
system["conduct_time_series_aggregation"] = True

system["optimized_years"] = 3
system["interval_between_years"] = 2
system["use_rolling_horizon"] = False
system["years_in_rolling_horizon"] = 1
```

- **scenarios.py:** To specify the individual scenarios of a multi-scenario simulation, this file is used similarly as in *test_6a*. By defining additional input data files, individual parameter values can be modified with respect to the default dataset, thus allowing a more efficient way than running a completely new dataset (Scenario Analysis).

optimization_setup.py

The *OptimizationSetup* class defines the optimization model by saving the properties of the *analysis* and *system* dictionaries. Using this information, the class adds the specified carriers and technologies to the optimization model such that it can be solved with its built-in solving method afterwards.

5.2.1. Objects

component.py

The *Component* class is used to add the parameters, variables and constraints to the optimization model represented by a linopy model. Since the linopy modelling language requires specific ways of how these parameters, variables and constraints must be constructed to suit its properties, *component.py* is needed. Component.py is therefore needed to adapt the parameters constructed via element.py such that they can be added to the linopy optimization model.

element.py

The *Element* class serves as the incremental building block of all technologies and carriers by defining all the necessary methods to define specific carriers and technologies. Therefore, it is the parent class of the *Carrier* and the *Technology* class.

energy_system.py

The *EnergySystem* class contains methods to add parameters, variables and constraints, to the optimization problem. In general, these components concern system-wide properties such as carbon or cost metrics. Additionally, the connections between the individual nodes are calculated and the objective function is created and passed to the concrete model.

time_steps.py

Contains a helper class containing methods to deal with timesteps.

5.2.1.1. Carrier

carrier.py

This script defines the class *Carrier*, which describes the individual energy carriers such as electricity, heat, biomass, etc. By extracting the corresponding input files, the carrier-related parameters, variables, and constraints are created.

5.2.1.2. Technology

technology.py

Contains the *Technology* class which defines all the technology-related parameters, variables and constraints that hold for all the existing technologies.

conversion_technology.py

Creates parameters, variables and constraints specifically needed for conversion technologies (e.g., natural gas boiler).

storage_technology.py

Creates parameters, variables and constraints specifically needed for storage technologies (e.g., pumped hydro storage).

transport_technology.py

Creates parameters, variables and constraints specifically needed for transport technologies (e.g., power lines).

5.3. Postprocess

5.3.1. postprocess.py

The *Postprocess* class saves all the information contained in the optimization problem and all the system configurations such that the gathered solution and the whole setup can be accessed without running the optimization again.

5.3.2. results.py

The “Results” class can read the files created with the Postprocess class and contains methods to visualize these results.

6. Appendices

6.1. Config Settings

System

system[Dictionary Key] = Value Type(Value Options)
e.g.:

```
system['set_transport_technologies'] = ['power_line','natural_gas_pipeline','carbon_pipeline']
```

Dictionary Key:	Value Options/Meaning:	Value Type:
“set_conversion_technologies”	Subset of the names of the folders located in the *set_conversion_technologies* directory of the dataset	List of strings
“set_storage_technologies”	Subset of the names of the folders located in the *set_storage_technologies* directory of the dataset	List of strings
“set_transport_technologies”	Subset of the names of the folders located in the *set_transport_technologies* directory of the dataset	List of strings
“set_nodes”	Nodes defined in *set_nodes.csv*	List of strings
“reference_year”	Starting year of optimization	Positive integer >=1900
“unaggregated_time_steps_per_year”	First x hours of year (normally 8760 for full year)	Positive integer
“aggregated_time_steps_per_year”	Number of representative time steps for time series aggregation	Positive integer
“conduct_time_series_aggregation”	True/False	Bool (default: False)
“optimized_years”	Number of optimized years	Positive integer
“interval_between_years”	Every “i-th” year is optimized (e.g. every second year for interval of 2)	Positive integer
“use_rolling_horizon”	Switch between perfect- and myopic foresight	Bool (default: False)
“years_in_rolling_horizon”	Number of years in foresight horizon	Positive integer
“double_capex_transport”	Account transport technology capex twice: per installed capacity (€/W) and per installed capacity per length (€/(W*m))	Bool (default: False)

"exclude_parameter_from_TSA"	Choose if parameters defined in the optional file *exclude_parameter_from_TSA.csv* should be excluded from TSA	Bool (default: True)
"conduct_scenario_analysis"	Choose if scenario analysis should be conducted	bool (default: False)
"run_default_scenario"	Choose if default scenario should be executed or not	Bool (default: True)
"clean_sub_scenarios"	Choose if result files of scenarios that are not in the current scenario dict should be deleted	Bool (default: False)
"knowledge_depreciation_rate"	Rate at which knowledge stock of existing capacities is depreciated annually	Positive float ϵ [0,1]

Analysis

Dictionary Key:	Value Options/Meaning:	Value Type:
"objective"	"total_cost" or "total_carbon_emissions"	String
"sense"	"minimize" or "maximize"	String
"time_series_aggregation"	See tsam package	Dictionary
"folder_output"	Specify path of output folder to store optimization results (e.g. "./outputs/")	String
"output_format"	"h5", "json" or "gzip"	String
"max_output_size_mb"	Limit the maximum file size of json result files	Positive Integer (default: 500)
"use_capacities_existing"	Choose if existing capacity should be considered (brownfield vs. greenfield)	Bool (default: False)

Solver

Dictionary Key:	Value Options/Meaning:	Value Type:
"name"	Choose solver: "glpk" or "gurobi"	String (default: "glpk")
"solver_options"	Additional solver options (see *default_config.py*)	Dictionary
"keep_files"	Whether to keep temporary solver files	Bool (default: False)
"io_api"	Solver option: See linopy package	String
"analyze_numerics"	Whether details about the optimization problem's numerics should be printed in the console	Bool (default: False)
"recommend_base_units"	Check if there is a better set of base units in terms of the optimizations numerics	Bool (default: False)

"immutable_unit"	Define base units which must not be changed to find a better set of base units	List of strings
"range_of_exponents"	Range of exponents the base units are allowed to be shifted to become "better" e.g. {"min": -2, "max": 3} →base units can be decreased by 10^-2 or increased by 10^3	Dictionary (default: {"min": -3, "max": 3}
"define_ton_as_metric_ton"	Whether the unit "ton" should be a metric ton (True) or an imperial ton (False)	Bool (default: True)
"rounding_decimal_points"	Rounding tolerance for new capacity and unit multipliers	Positive integer (default: 5)
"rounding_decimal_points_ts"	Rounding tolerance for time series after TSA	Positive integer (default: 5)
"linear_regression_check"	Tolerances to determine if PWA-modelled x-y relationships can be represented by a linear slope while meeting these tolerances	Dictionary (default: {"eps_intercept": 0.1, "epsRvalue": 1-1E-5}

6.2. Parameter, Variable and Constraint Overview

Parameters			
Name:	**Time Step Type:**	**Doc String:**	**Scope:**
carbon_emissions_limit	set_time_steps_yearly	Parameter which specifies the total limit on carbon emissions	system
carbon_emissions_budget	temporal immutable	Parameter which specifies the total budget of carbon emissions until the end of the entire time horizon	system
carbon_emissions_cumulative_existing	temporal immutable	Parameter which specifies the total previous carbon emissions	system
price_carbon_emissions	set_time_steps_yearly	Parameter which specifies the yearly carbon price	system
price_carbon_emissions_overshoot	temporal immutable	Parameter which specifies the carbon price for budget overshoot	system
market_share_unbounded	temporal immutable	Parameter which specifies the unbounded market share	system
knowledge_spillover_rate	temporal immutable	Parameter which specifies the knowledge spillover rate	system
time_steps_operation_duration	set_time_steps_operation	Parameter which specifies the time step duration in operation for all technologies	system
demand	set_time_steps_operation	Parameter which specifies the carrier demand	carrier
availability_import	set_time_steps_operation	Parameter which specifies the maximum energy that can be imported from outside the system boundaries	carrier
availability_export	set_time_steps_operation	Parameter which specifies the maximum energy that can be exported to outside the system boundaries	carrier
availability_import_yearly	set_time_steps_yearly	Parameter which specifies the maximum energy that can be imported from outside the system boundaries for the entire year	carrier
availability_export_yearly	set_time_steps_yearly	Parameter which specifies the maximum energy that can be exported to outside the system boundaries for the entire year	carrier

price_import	set_time_steps_operation	Parameter which specifies the import carrier price	carrier
price_export	set_time_steps_operation	Parameter which specifies the export carrier price	carrier
price_shed_demand	temporal immutable	Parameter which specifies the price to shed demand	carrier
carbon_intensity_carrier	set_time_steps_yearly	Parameter which specifies the carbon intensity of carrier	carrier
capacity_existing	temporal immutable	Parameter which specifies the existing technology size	technology
capacity_investment_existing	set_time_steps_yearly_entire_horizon	Parameter which specifies the size of the previously invested capacities	technology
capacity_addition_min	temporal immutable	Parameter which specifies the minimum capacity addition that can be installed	technology
capacity_addition_max	temporal immutable	Parameter which specifies the maximum capacity addition that can be installed	technology
capacity_addition_unbounded	temporal immutable	Parameter which specifies the unbounded capacity addition that can be added each year (only for delayed technology deployment)	technology
lifetime_existing	temporal immutable	Parameter which specifies the remaining lifetime of an existing technology	technology
capex_capacity_existing	temporal immutable	Parameter which specifies the total capex of an existing technology which still has to be paid	technology
opex_specific_variable	set_time_steps_operation	Parameter which specifies the variable specific opex	technology
opex_specific_fixed	set_time_steps_yearly	Parameter which specifies the fixed annual specific opex	technology
lifetime	temporal immutable	Parameter which specifies the lifetime of a newly built technology	technology
construction_time	temporal immutable	Parameter which specifies the construction time of a newly built technology	technology
max_diffusion_rate	set_time_steps_yearly	Parameter which specifies the maximum diffusion rate which is the maximum increase in capacity between investment steps	technology
capacity_limit	temporal immutable	Parameter which specifies the capacity limit of technologies	technology

min_load	set_time_steps_operation	Parameter which specifies the minimum load of technology relative to installed capacity	technology
max_load	set_time_steps_operation	Parameter which specifies the maximum load of technology relative to installed capacity	technology
carbon_intensity_technology	temporal immutable	Parameter which specifies the carbon intensity of each technology	technology
capex_specific_conversion	set_time_steps_yearly	Parameter which specifies the slope of the capex if approximated linearly	conversion technology
conversion_factor	set_time_steps_yearly	Parameter which specifies the slope of the conversion efficiency if approximated linearly	conversion technology
time_steps_storage_level_duration	set_time_steps_storage_level	Parameter which specifies the time step duration in StorageLevel for all technologies	storage technology
efficiency_charge	set_time_steps_yearly	efficiency during charging for storage technologies	storage technology
efficiency_discharge	set_time_steps_yearly	efficiency during discharging for storage technologies	storage technology
self_discharge	temporal immutable	self-discharge of storage technologies	storage technology
capex_specific_storage	set_time_steps_yearly	specific capex of storage technologies	storage technology
distance	temporal immutable	distance between two nodes for transport technologies	transport technology
capex_specific_transport	set_time_steps_yearly	capex per unit for transport technologies	transport technology
capex_per_distance_transport	set_time_steps_yearly	capex per distance for transport technologies	transport technology
transport_loss_factor	temporal immutable	carrier losses due to transport with transport technologies	transport technology

Variables			
Name:	**Time Step Type:**	**Doc String:**	**Scope:**
carbon_emissions_total	set_time_steps_yearly	total carbon emissions of energy system	system
carbon_emissions_cumulative	set_time_steps_yearly	cumulative carbon emissions of energy system over time for each year	system
carbon_emissions_overshoot	set_time_steps_yearly	overshoot carbon emissions of energy system at the end of the time horizon	system
cost_carbon_emissions_total	set_time_steps_yearly	total cost of carbon emissions of energy system	system
cost_total	set_time_steps_yearly	total cost of energy system	system
net_present_cost	set_time_steps_yearly	net_present_cost of energy system	system
flow_import	set_time_steps_operation	node- and time-dependent carrier import from the grid	carrier
flow_export	set_time_steps_operation	node- and time-dependent carrier export from the grid	carrier
cost_carrier	set_time_steps_operation	node- and time-dependent carrier cost due to import and export	carrier
cost_carrier_total	set_time_steps_yearly	total carrier cost due to import and export	carrier
carbon_emissions_carrier	set_time_steps_operation	carbon emissions of importing and exporting carrier	carrier
carbon_emissions_carrier_total	set_time_steps_yearly	total carbon emissions of importing and exporting carrier	carrier
shed_demand	set_time_steps_operation	shed demand of carrier	carrier
cost_shed_demand	set_time_steps_operation	shed demand of carrier	carrier
technology_installation	set_time_steps_yearly	installment of a technology at location l and time t	technology
capacity	set_time_steps_yearly	size of installed technology at location l and time t	technology
capacity_addition	set_time_steps_yearly	size of built technology (invested capacity after construction) at location l and time t	technology
capacity_investment	set_time_steps_yearly	size of invested technology at location l and time t	technology
cost_capex	set_time_steps_yearly	capex for building technology at location l and time t	technology
capex_yearly	set_time_steps_yearly	annual capex for having technology at location l	technology
cost_capex_total	set_time_steps_yearly	total capex for installing all technologies in all locations at all times	technology
cost_opex	set_time_steps_operation	opex for operating technology at location l and time t	technology
opex_yearly	set_time_steps_yearly	yearly opex for operating technology at location l and year y	technology
cost_opex_total	set_time_steps_yearly	total opex all technologies and locations in year y	technology
carbon_emissions_technology	set_time_steps_operation	carbon emissions for operating technology at location l and time t	technology

carbon_emissions_technology_total	set_time_steps_yearly	total carbon emissions for operating technology at location l and time t	technology
flow_conversion_input	set_time_steps_operation	Carrier input of conversion technologies	conversion technology
flow_conversion_output	set_time_steps_operation	Carrier output of conversion technologies	conversion technology
capacity_approximation	set_time_steps_yearly	pwa variable for size of installed technology on edge i and time t	technology
capex_approximation	set_time_steps_yearly	pwa variable for capex for installing technology on edge i and time t	technology
flow_approximation_reference	set_time_steps_operation	pwa of flow of reference carrier of conversion technologies	conversion technology
flow_approximation_dependent	set_time_steps_operation	pwa of flow of dependent carriers of conversion technologies	conversion technology
flow_storage_charge	set_time_steps_operation	carrier flow into storage technology on node i and time t	storage technology
flow_storage_discharge	set_time_steps_operation	carrier flow out of storage technology on node i and time t	storage technology
storage_level	set_time_steps_storage_level	storage level of storage technology on node in each storage time step	storage technology
flow_transport	set_time_steps_operation	carrier flow through transport technology on edge i and time t	transport technology
flow_transport_loss	set_time_steps_operation	carrier flow through transport technology on edge i and time t	transport technology

Constraints			
Name:	**Time Step Type:**	**Doc String:**	**Scope:**
constraint_carbon_emissions_total	set_time_steps_yearly	total carbon emissions of energy system	system
constraint_carbon_emissions_cumulative	set_time_steps_yearly	cumulative carbon emissions of energy system over time	system
constraint_carbon_cost_total	set_time_steps_yearly	total carbon cost of energy system	system
constraint_carbon_emissions_limit	set_time_steps_yearly	limit of total carbon emissions of energy system	system
constraint_carbon_emissions_budget	set_time_steps_yearly	Budget of total carbon emissions of energy system	system

constraint_carbon_emissions_overshoot_limit	set_time_steps_yearly	Limit of overshot carbon emissions of energy system	system
constraint_cost_total	set_time_steps_yearly	total cost of energy system	system
constraint_net_present_cost	set_time_steps_yearly	net_present_cost of energy system	system
constraint_availability_import	set_time_steps_operation	node- and time-dependent carrier availability to import from outside the system boundaries	carrier
constraint_availability_export	set_time_steps_operation	node- and time-dependent carrier availability to export to outside the system boundaries	carrier
constraint_availability_import_yearly	set_time_steps_yearly	node- and time-dependent carrier availability to import from outside the system boundaries summed over entire year	carrier
constraint_availability_export_yearly	set_time_steps_yearly	node- and time-dependent carrier availability to export to outside the system boundaries summed over entire year	carrier
constraint_cost_carrier	set_time_steps_operation	cost of importing and exporting carrier	carrier
constraint_cost_shed_demand	set_time_steps_operation	cost of shedding carrier demand	carrier
constraint_limit_shed_demand	set_time_steps_operation	limit of shedding carrier demand	carrier
constraint_cost_carrier_total	set_time_steps_yearly	total cost of importing and exporting carriers	carrier
constraint_carbon_emissions_carrier	set_time_steps_operation	carbon emissions of importing and exporting carrier	carrier
constraint_carbon_emissions_carrier_total	set_time_steps_yearly	total carbon emissions of importing and exporting carriers	carrier
constraint_nodal_energy_balance	set_time_steps_operation	node- and time-dependent energy balance for each carrier	carrier
constraint_technology_capacity_limit	set_time_steps_yearly	limited capacity of technology depending on loc and time	technology
constraint_technology_min_capacity	set_time_steps_yearly	min capacity of technology that can be installed	technology
constraint_technology_max_capacity	set_time_steps_yearly	max capacity of technology that can be installed	technology
constraint_technology_construction_time	set_time_steps_yearly	lead time in which invested technology is constructed	technology
constraint_technology_lifetime	set_time_steps_yearly	max capacity of technology that can be installed	technology
constraint_technology_diffusion_limit	set_time_steps_yearly	Limits the newly built capacity by the existing knowledge stock	technology
constraint_capacity_factor	set_time_steps_operation	limit max load by installed capacity	technology

constraint_capex_yearly	set_time_steps_yearly	annual capex of having capacity of technology.	technology
constraint_cost_capex_total	set_time_steps_yearly	total capex of all technology that can be installed.	technology
constraint_opex_technology	set_time_steps_operation	opex for each technology at each location and time step	technology
constraint_opex_yearly	set_time_steps_yearly	total opex of all technology that are operated.	technology
constraint_cost_opex_total	set_time_steps_yearly	total opex of all technology that are operated.	technology
constraint_carbon_emissions_technology	set_time_steps_operation	carbon emissions for each technology at each location and time step	technology
constraint_carbon_emissions_technology_total	set_time_steps_yearly	total carbon emissions for each technology at each location and time step	technology
disjunct_on_technology	set_time_steps_operation	disjunct to indicate that technology is on	technology
disjunct_off_technology	set_time_steps_operation	disjunct to indicate that technology is off	technology
disjunction_decision_on_off_technology	set_time_steps_operation	disjunction to link the on off disjuncts	technology
constraint_linear_capex	set_time_steps_yearly	Linear relationship in capex	technology
constraint_linear_conversion_factor	set_time_steps_operation	Linear relationship in conversion_factor	conversion technology
constraint_capex_coupling	set_time_steps_yearly	couples the real capex variables with the approximated variables	conversion technology
constraint_capacity_coupling	set_time_steps_yearly	couples the real capacity variables with the approximated variables	conversion technology
constraint_reference_flow_coupling	set_time_steps_operation	couples the real reference flow variables with the approximated variables	conversion technology
constraint_dependent_flow_coupling	set_time_steps_operation	couples the real dependent flow variables with the approximated variables	conversion technology
constraint_storage_level_max	set_time_steps_storage_level	limit maximum storage level to capacity	storage technology
constraint_couple_storage_level	set_time_steps_storage_level	couple subsequent storage levels (time coupling constraints)	storage technology
constraint_storage_technology_capex	set_time_steps_yearly	Capital expenditures for installing storage technology	storage technology
constraint_transport_technology_losses_flow	set_time_steps_operation	Carrier loss due to transport with through transport technology	transport technology

constraint_transport_technology_capex	set_time_steps_yearly	Capital expenditures for installing transport technology	transport technology
constraint_transport_technology_bidirectional	set_time_steps_yearly	Forces that transport technology capacity must be equal in both directions	transport technology