deflex Documentation

Release 0.4.0

CC BY 4.0 - Uwe Krien

Jun 22, 2022

Contents

CHAPTER 1

deflex - flexible multi-regional energy system model for heat, power and mobility

++++++ multi sectoral energy system of Germany/Europe ++++++ dispatch optimisation ++++++ highly configurable and adaptable ++++++ multiple analyses functions +++++

The following README gives you a brief overview about deflex. Read the full documentation for all information.

1.1 Installation

To run *deflex* you have to install the Python package and a solver:

- deflex is available on PyPi and can be installed using `pip install deflex`.
- an LP-solver is needed such as CBC (default), GLPK, Gurobi*, Cplex*
- for some extra functions additional packages and are needed

* Proprietary solver

1.2 Examples

1. Run `pip install deflex[example]` to get all dependencies.
2. Create a local directory (e.g. /home/user/deflex_examples).
3. Browse the examples for deflex v0.4.x or download all examples as zip file and copy/extract them to your local directory.
4. Read the comments of each example, execute it and modify it to your needs. Do not forget to set a local path in the examples if needed.
5. In parallel you should read the `usage guide` of the documentation to get the full picture.

The example scripts will download the example scenarios to the $HOME/deflex folder. It is also possible to browse the example scenarios.

1.3 Improve deflex

We are warmly welcoming all who want to contribute to the deflex library. This includes the following actions:

- Write bug reports or comments
- Improve the documentation (including typos, grammar)
- Add features improve the code (open an issue first)

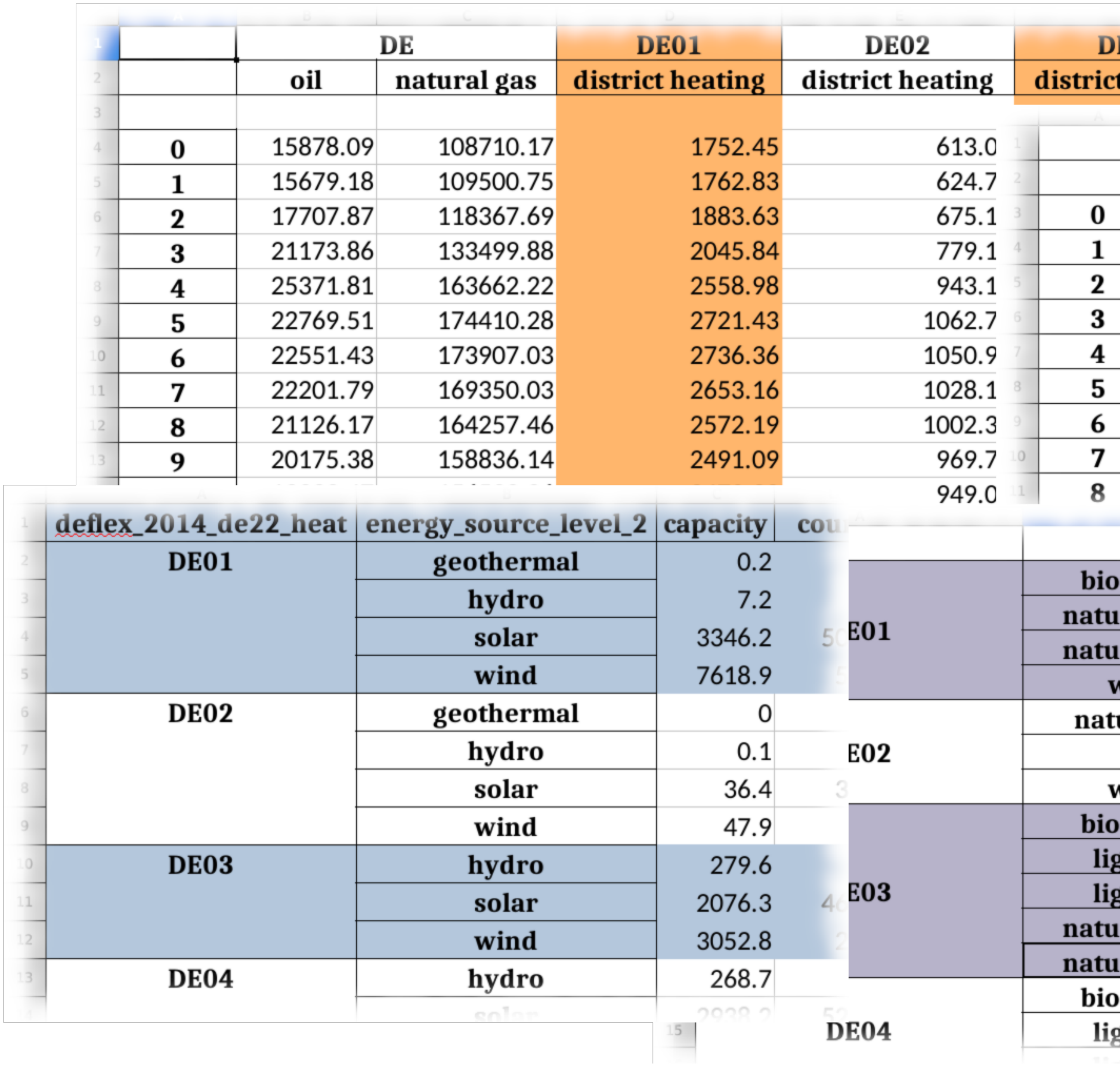
1.4 Citing deflex

Go to the Zenodo page of deflex to find the DOI of your version. To cite all deflex versions use:

1.5 Gallery

The following figures will give you a brief impression about deflex.

Figure 1: Use one of the include regions sets or create your own one. You can also include other European countries.



	DE		DE01	DE02	DI
	oil	natural gas	district heating	district heating	district
0	15878.09	108710.17	1752.45	613.0	
1	15679.18	109500.75	1762.83	624.7	
2	17707.87	118367.69	1883.63	675.1	
3	21173.86	133499.88	2045.84	779.1	
4	25371.81	163662.22	2558.98	943.1	
5	22769.51	174410.28	2721.43	1062.7	
6	22551.43	173907.03	2736.36	1050.9	
7	22201.79	169350.03	2653.16	1028.1	
8	21126.17	164257.46	2572.19	1002.3	
9	20175.38	158836.14	2491.09	969.7	
				949.0	

deflex_2014_de22_heat	energy_source_level_2	capacity	cou
DE01	geothermal	0.2	
	hydro	7.2	
	solar	3346.2	5
	wind	7618.9	
DE02	geothermal	0	
	hydro	0.1	
	solar	36.4	3
	wind	47.9	
DE03	hydro	279.6	
	solar	2076.3	4
	wind	3052.8	2
DE04	hydro	268.7	

	0
	1
	2
	3
	4
	5
	6
	7
	8

E01	bio
	natu
	natu
	v
E02	natu
	v
E03	bio
	lig
	lig
	natu
	natu
	bio
DE04	lig

Figure 2: The input data can be organised in spreadsheets or csv files.

Figure 3: The resulting system costs of deflex have been compared with the day-ahead prices from the Entso-e downloaded from Open Power System Data. The plot shows three different periods of the year.

Figure 4: It is also possible to get a time series of the average emissions. Furthermore, it shows the emissions of the most expensive power plant which would be replaced by an additional feed-in.

Figure 5: The following plot shows fraction of the time on which the utilisation of the power lines between the regions is more than 90% of its maximum capacity:

1.6 Documentation

The full documentation of deflex is available on readthedocs.

Go to the download page to download different versions and formats (pdf, html, epub) of the documentation.

1.7 License

CHAPTER 2

Installation guide

The deflex package is available on PyPi.

2.1 Basic version

The basic version of deflex can read, solve and analyse a deflex scenario. Some additional functions such as spatial operations or plots need some extra packages (see below). To install the latest stable version use:

```
pip install deflex
```

In case you have some old deflex scenario you can install the *old stable phd version*:

```
pip install https://github.com/reegis/deflex/archive/phd.zip
```

To get the latest features you can install the *testing version*:

```
pip install https://github.com/reegis/deflex/archive/master.zip
```

2.2 Installation of a solver (mandatory)

To solve an energy system a linear solver has to be installed. For the communication with the solver *Pyomo* is used. Have a look at the Pyomo docs to learn about which solvers are supported.

The default solver for deflex is the CBC solver. Go to the oemof.solph documentation to get help for the solver installation.

2.3 Additional requirements (optional)

The basic installation can be used to compute scenarios (csv, xls, xlsx). For some functions additional packages are needed. Some of these packages may need OS specific packages. Please see the installation guide of each package if an error occur.

1. **To run the example with plots you need the following packages:**

    - matplotlib (plotting)
    - pytz (time zones)
    - requests (download example files)

    ```
    pip install deflex[example]
    ```

2. **To use the maps of the polygons, transmission lines etc.:**

    - pygeos (spatial operations)
    - geopandas (maps)

    ```
    pip install deflex[map]
    ```

3. **To develop deflex:**

    - pytest
    - sphinx
    - sphinx_rtd_theme
    - pygeos
    - geopandas
    - requests

    ```
    pip install deflex[dev]
    ```

CHAPTER 3

Usage guide

THIS CHAPTER IS WORK IN PROGRESS…

3.1 DeflexScenario

The scenario class `DeflexScenario` is a central element of deflex.

All input data is stored as a dictionary in the `input_data` attribute of the `DeflexScenario` class. The keys of the `dictionary` are names of the data table and the values are `pandas.DataFrame` or `pandas.Series` with the data.

3.1.1 Load input data

At the moment, there are two methods to populate this attribute from files:

- read_csv() - read a directory with all needed csv files.
- read_xlsx() - read a spread sheet in the `.xlsx`

To learn how to create a valid input data set see “REFERENCE”.

```
from deflex import scenario
sc = scenario.DeflexScenario()
sc.read_xlsx("path/to/xlsx/file.xlsx")
# OR
sc.read_csv("path/to/csv/dir")
```

3.1.2 Solve the energy system

A valid input data set describes an energy system. To optimise the dispatch of the energy system a external solver is needed. By default the CBC solver is used but different solver are possible (see: solver).

The simplest way to solve a scenario is the `compute()` method.

```
sc.compute()
```

To use a different solver one can pass the `solver` parameter.

```
sc.compute(solver="glpk")
```

3.1.3 Store and restore the scenario

The `dump()` method can be used to store the scenario. a solved scenario will be stored with the results. The scenario is stored in a binary format and it is not human readable.

```
sc.dump("path/to/store/results.dflx")
```

To restore the scenario use the `restore_scenario` function:

```
sc = scenario.restore_scenario("path/to/store/results.dflx")
```

3.1.4 Analyse the scenario

Most analyses cannot be taken if the scenario is not solved. However, the merit order can be shown only based on the input data:

```
from deflex import DeflexScenario
from deflex import analyses
sc = DeflexScenario()
sc.read_xlsx("path/to/xlsx/file.xlsx")
pp = analyses.merit_order_from_scenario(sc)
ax = plt.figure(figsize=(15, 4)).add_subplot(1, 1, 1)
ax.step(pp["capacity_cum"].values, pp["costs_total"].values, where="pre")
ax.set_xlabel("Cumulative capacity [GW]")
ax.set_ylabel("Marginal costs [EUR/MWh]")
ax.set_ylim(0)
ax.set_xlim(0, pp["capacity_cum"].max())
plt.show()
```

With the *de02_co2-price_var-costs.xlsx* from the examples the code above will produce the following plot:

Filling the area between the line and the x-axis with colors according the fuel of the power plant oen get the following plot:

IMPORTANT: This is just an example and not a source for the actual merit order in Germany.

3.2 Scripts

3.2.1 Console scripts

The console scripts can be used to model a scenario without using Python directly (Python need to be installed, though).

Get the help message by typing

```
deflex-compute --help
```

The reference can also be found here: *main()*

If you dumped your results you can also use postprocessing tools. Read the help message for more information by typing:

```
deflex-result --help
```

The reference can also be found here: `result()` See the results section for more information about the postprocessing function and classes. Not all postprocessing tools can be used with the console script.

3.2.2 Python scripts

For the typical work flow (creating a scenario, loading the input data, computing the scenario and storing the results) the `model_scenario()` function can be used.

To collect all scenarios from a given directory the function `search_input_scenarios()` can be used. The function will search for `.xlsx` files or paths that end on `_csv` and cannot distinguish between a valid scenario and any `.xlsx` file or paths that accidentally contain `_csv`.

No matter how you collect a list of a scenario input data files the `batch_model_scenario()` function makes it easier to run each scenario and get back the relevant information about the run. It is possible to ignore exceptions so that the script will go on with the following scenarios if one scenario fails.

If you have enough memory and cpu capacity on your computer/server you can optimise your scenarios in parallel. Use the `model_multi_scenarios()` function for this task. You can pass a list of scenario files to this function. A cpu fraction will limit the number of processes as a fraction of the maximal available number of cpu cores. Keep in mind that for large models the memory will be the limit not the cpu capacity. If a memory error occurs the script will stop immediately. It is not possible to catch a memory error. A log-file will log all failing and successful runs.

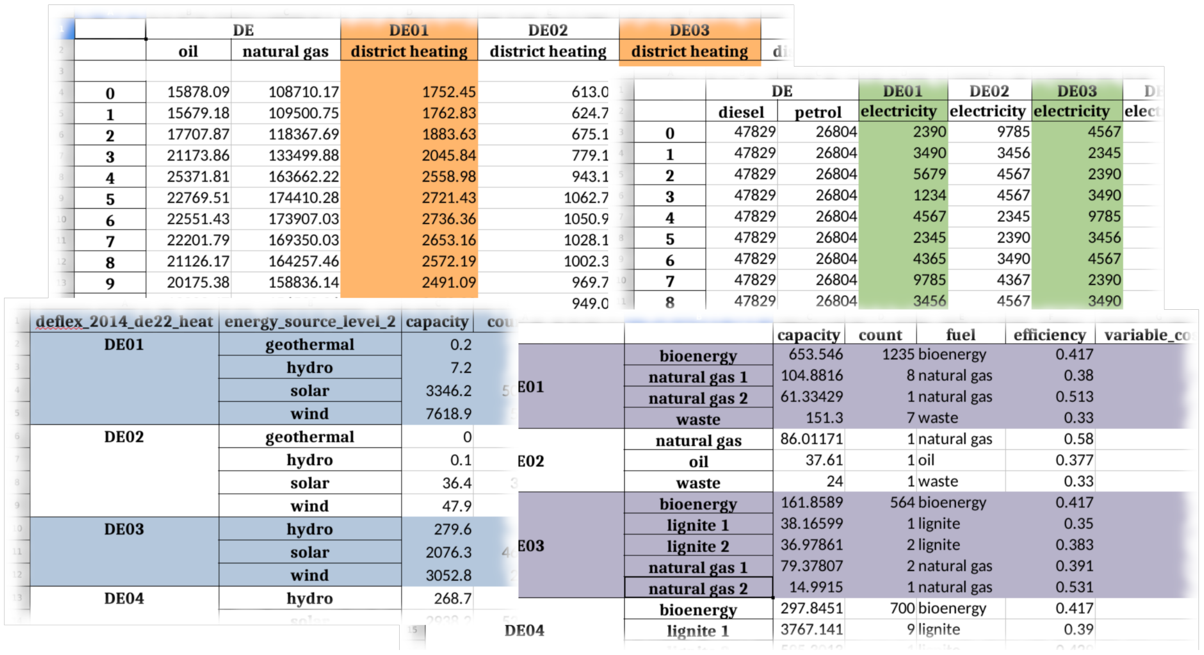
3.3 Input data

The input data is stored in the *input_data* attribute of the `DeflexScenario` class (s. *DeflexScenario*). It is a dictionary with the name of the data set as key and the data table itself as value (pandas.DataFrame or pandas.Series).

The input data is divided into four main topics: High-level-inputs, electricity sector, heating sector (optional) and mobility sector (optional).

Download examples (link) to get an idea of the typical structure. Then go on with the following chapter to learn everything about how to define the data of a deflex model.

3.3.1 Overview



	DE		DE01	DE02	DE03	di
	oil	natural gas	district heating	district heating	district heating	
0	15878.09	108710.17	1752.45	613.0		
1	15679.18	109500.75	1762.83	624.7		
2	17707.87	118367.69	1883.63	675.1		
3	21173.86	133499.88	2045.84	779.1		
4	25371.81	163662.22	2558.98	943.1		
5	22769.51	174410.28	2721.43	1062.7		
6	22551.43	173907.03	2736.36	1050.9		
7	22201.79	169350.03	2653.16	1028.1		
8	21126.17	164257.46	2572.19	1002.3		
9	20175.38	158836.14	2491.09	969.7		
				949.0		

	DE		DE01	DE02	DE03	D
	diesel	petrol	electricity	electricity	electricity	elec
0	47829	26804	2390	9785	4567	
1	47829	26804	3490	3456	2345	
2	47829	26804	5679	4567	2390	
3	47829	26804	1234	4567	3490	
4	47829	26804	4567	2345	9785	
5	47829	26804	2345	2390	3456	
6	47829	26804	4365	3490	4567	
7	47829	26804	9785	4367	2390	
8	47829	26804	3456	4567	3490	

deflex_2014_de22_heat	energy_source_level_2	capacity	cou
DE01	geothermal	0.2	
	hydro	7.2	
	solar	3346.2	
	wind	7618.9	
DE02	geothermal	0	
	hydro	0.1	
	solar	36.4	
	wind	47.9	
DE03	hydro	279.6	
	solar	2076.3	
	wind	3052.8	
DE04	hydro	268.7	

		capacity	count	fuel	efficiency	variable_co
E01	bioenergy	653.546	1235	bioenergy	0.417	
	natural gas 1	104.8816	8	natural gas	0.38	
	natural gas 2	61.33429	1	natural gas	0.513	
	waste	151.3	7	waste	0.33	
E02	natural gas	86.01171	1	natural gas	0.58	
	oil	37.61	1	oil	0.377	
	waste	24	1	waste	0.33	
E03	bioenergy	161.8589	564	bioenergy	0.417	
	lignite 1	38.16599	1	lignite	0.35	
	lignite 2	36.97861	2	lignite	0.383	
	natural gas 1	79.37807	2	natural gas	0.391	
	natural gas 2	14.9915	1	natural gas	0.531	
DE04	bioenergy	297.8451	700	bioenergy	0.417	
	lignite 1	3767.141	9	lignite	0.39	

A Deflex scenario can be divided into regions. Each region must have an identifier number and be named after it as `DEXX`, where `XX` is the number. For refering the Deflex scenario as a whole (i.e. the sum of all regions) use `DE` only.

At the current state the distribution of fossil fuels is neglected. Therefore, in order to keep the computing time low it is recommended to define them supra-regional using `DE` without a number. It is still possible to define them regional for example to add a specific limit for each region.

Note: The nomenclature above is the one used in the examples. It is also possible to extend it e.g. for surrounding countries (`AT`, `FR`, `PL...`) or to totally deviate from it. Nevertheless, it might be helpful to keep the basic idea of using the country code of the top level domain followed by a number if subregions exist or without a number. This will help other users to understand your data.

In most cases it is also sufficient to model the fossil part of the mobility and the decentralised heating sector supra-regional. It is assumed that a gas boiler or a filling station is always supplied with enough fuel, so that only the annual values affect the model. This does not apply to electrical heating systems or cars.

In most spread sheet software it is possible to connect cells to increase readability. These lines are interpreted correctly. In csv files the values have to appear in every cell. So the following two tables will be interpreted equally!

Connected cells

		value
DE01	F1	
	F2	
DE02	F1	

Unconnected cells

		value
DE01	F1	
DE01	F2	
DE02	F1	

Note: NaN-values are not allowed in any table. Some columns are optional and can be left out, but if a column is present there have to be values in every row. Neutral values can be `0`, `1` or `inf`.

3.3.2 High-level-input (mandatory)

General

`key:` 'general', `value:` pandas.Series()

This table contains basic data about the scenario.

year	
co2 price	
number of time steps	
name	

INDEX

year: `int`, [-] A time index will be created starting with January 1, at 00:00 with the number of hours given in *number of time steps*.

co2 price: `float`, [€/t] The average price for CO_2 over the whole time period.

number of time steps: `int`, [-] The number of hourly time steps.

name: `str`, [-] A name for the scenario. This name will be used to compare key values between different scenarios. Therefore, it should be unique within a group of scenarios. It does not have to be intuitive. Use the *info* table for a human readable description of your scenario.

Info

`key:` 'info', `value:` pandas.Series()

On this sheet, additional information that characterizes the scenario can be added. The idea behind Info is that the user can filter stored scenarios using the `search_dumped_scenarios()` function.

You can create any key-value pair which is suitable for a group of scenarios.

e.g. key: `scenario_type` value: `foo` / `bar` / `foobar`

Afterwards you can search for all scenarios where the `scenario_type` is `foo` using `search_dumped_scenarios()`. See documentation and examples of this function for more details.

key1	
key2	
key3	
…	…

Commodity sources

`key`: 'commodity sources', `value`: pandas.DataFrame()

This sheet requires data from all the commodities used in the scenario. The data can be provided either supra-regional under DE, regional under DEXX or as a combination of both, where some commodities are global and some are regional. Regionalised commodities are especially useful for commodities with an annual limit, for example bioenergy.

		costs	emission	annual limit
DE	F1			
	F2			
DE01	F1			
DE02	F2			
…	…	…	…	…

INDEX

level 0: `str` Region (e.g. DE01, DE02 or DE).

level 1: `str` Fuel type (e.g. natural gas or bionenergy).

COLUMNS

costs: `float`, [€/MWh] The fuel production cost.

emission: `float`, [t/MWh] The fuel emission factor.

annual limit: `float`, [MWh] The annual maximum energy generation (if there is one, otherwise just use *inf*). If the `annual limit` is `inf` in every line the column can be left out.

Data sources

`key`: 'data sources', `value`: pandas.DataFrame()

Highly recomended. Here the type data, the source name and the url from where they were obtained can be listed. It is a free format and additional columns can be added. This table helps to make your scenario as transparent as possible.

	source	url	v1	…
cost data	Institute	http1	a1	…
pv plants	Organisation	http2	a2	…
…	…	…	…	…

3.3.3 Electricity sector (mandatory)

Electricity demand series

`key:` 'electricity demand series', `value:` pandas.DataFrame()

This sheet requires the electricity demand of the scenario as a time series. One summarised demand series for each region is enough, but it is possible to distinguish between different types. This will not have any effect on the model results but may help to distinguish the different flows in the results.

	DE01	DE02			DE03	…
	all	industry	buildings	rest	all	…
Time step 1						…
Time step 2						…
…	…	…	…	…	…	…

INDEX

time step: `int` Number of time step. Must be uniform in all series tables.

COLUMNS

unit: `[MW]`

level 0: `str` Region (e.g. DE01, DE02).

level 1: `str` Specification of the series e.g. "all" for an overall series.

Power plants

`key:` 'power plants', `value:` pandas.DataFrame()

The power plants will feed in the electricity bus of the region the are located. The data must be divided by region and subdivided by fuel. Each row can indicate one power plant or a group of power plants. It is possible to add additional columns for information purposes.

		capacity	fuel	efficiency	annual electricity limit	variable_cost	downtime_factor	source_region
DE01	N1							
	N2							
	N3							
DE02	N2							
	N3							
…	…	…	…	…	…	…	…	…

INDEX

level 0: `str` Region (e.g. DE01, DE02).

level 1: `str` Name, arbitrary. The combination of region and name is the unique identifier for the power plant or the group of power plants.

COLUMNS

capacity: `float`, [MW] The installed capacity of the power plant or the group of power plants.

fuel: `str`, [-] The used fuel of the power plant or group of power plants. The combination of *source_region* and *fuel* must exist in the commodity sources table.

efficiency: `float`, [-] The average overall efficiency of the power plant or the group of power plants.

annual limit: `float`, [MWh] The absolute maximum limit of produced electricity within the whole modeling period.

variable_costs: `float`, [€/MWh] The variable costs per produced electricity unit.

downtime_factor: `float`, [-] The time fraction of the modeling period in which the power plant or the group of power plants cannot produce electricity. The installed capacity will be reduced by this factor `capacity * (1 - downtime_factor)`.

source_region, [-] The source region of the fuel source. Typically this is the region of the index or `DE` if it is a global commodity source. The combination of *source_region* and *fuel* must exist in the commodity sources table.

Volatiles plants

`key:` 'volatile plants', `value:` pandas.DataFrame()

Examples of volatile power plants are solar, wind, hydro, geothermal. Data must be provided divided by region and subdivided by energy source. Each row can indicate one plant or a group of plants. It is possible to add additional columns for information purposes.

		capacity
DE01	N1	
	N2	
DE02	N1	
DE03	N1	
	N3	
…	…	…

INDEX

level 0: `str` Region (e.g. DE01, DE02).

level 1: `str` Name, arbitrary. The combination of the region and the name has to exist as a time series in the *volatile series* table.

COLUMNS

capacity: `float`, [MW] The installed capacity of the plant.

Volatile series

`key:` 'volatile series', `value:` pandas.DataFrame()

This sheet provides the normalised feed-in time series in MW/MW $_{\text{installed}}$. So each time series will multiplied with its installed capacity to get the absolute feed-in. Therefore, the combination of region and name has to exist in the *volatile plants* table.

	DE01		DE02	DE03		…
	N1	N2	N1	N1	N3	…
Time step 1						…
Time step 2						…
…	…	…	…	…	…	…

INDEX

time step: `int` Number of time step. Must be uniform in all series tables.

COLUMNS

unit: `[MW]`

level 0: `str` Region (e.g. DE01, DE02).

level 1: `str` Name of the energy source specified in the previous sheet.

Power lines

`key:` 'power lines', `value:` pandas.DataFrame()

The power lines table defines the connection between the electricity buses of each region of the scenario. There is no default connection. If no connection is defined the regions will be self-sufficient.

	capacity	efficiency
DE01-DE02		
DE01-DE03		
DE02-DE03		
…	…	…

INDEX

Name: `str` Name of the 2 connected regions separated by a dash. Define only one direction. In the model one line for each direction will be created. If both directions are defined in the table two lines for each direction will be created for the model, so that the capacity will be the sum of both lines.

COLUMNS

capacity: `float`, [MW] The maximum transmission capacity of the power lines.

efficiency:`float`, [-] The transmission efficiency of the power line.

Electricity storages

`key`: 'storages', `value`: pandas.DataFrame()

Electricity storages is a particular case of storages (see *Storages*). The condition to use a storage as an electricity storage is to define `electricity` in the `storage medium` column.

3.3.4 Heating sector (optional)

Heat demand series

`key`: 'heat demand series', `value`: pandas.DataFrame()

The heat demand can be entered regionally under DEXX or supra-regional under DE. The only type of demand that must be entered regionally is district heating. As recommendation, coal, gas, or oil demands should be treated supra-regional.

	DE01		DE02				DE		
	district heating	N1	district heating	N1	N2	…	N3	N4	N5
Time step 1									
Time step 2									
…	…	…	…	…	…	…	…	…	…

INDEX

time step: **`int`** Number of time step. Must be uniform in all series tables.

COLUMNS

unit: `[MW]`

level 0: **`str`** Region (e.g. DE01, DE02 or DE).

level 1: **`str`** Name. Specification of the series e.g. *district heating*, *coal*, *gas*. Except for *district heating* each combination of region and name must exist in the *decentralised heat* table.

Decentralised heat

`key`: 'decentralised heat', `value`: pandas.DataFrame()

This sheet covers all heating technologies that are used to generate decentralized heat. In this context decentralised does not mean regional it represents the large group of independent heating systems. If there is no specific reason to define a heating system regional they should be defined supra-regional.

		efficiency	source	source region
DE01	N1			DE01
DE02	N1			DE02
	N2			DE02
	…			…
DE	N3			DE
	N4			DE
	N5			DE

INDEX

level 0: `str` Region (e.g. DE01, DE02 or DE).

level 1: `str` Name, arbitrary.

COLUMNS

efficiency: `float`, [-] The efficiency of the heating technology.

source: `str`, [-] The source that the heating technology uses. Examples are coal, oil for commodities, but it could also be electricity in case of a heat pump. Except for *electricity* the combination of *source* and *source region* has to exist in the *commodity sources* table. The *electricity* source will be connected to the electricity bus of the region defined in *source region*.

source region: `str` The region where the source comes from (see *source*).

Chp - heat plants

`key:` 'chp-heat plants', `value:` pandas.DataFrame()

This sheet covers CHP and heat plants. Each plant will feed into the *district heating* bus of the region it it is located. The demand of *district heating* is defined in the *heat demand series* table with the name *district heating*. All plants of the same region with the same fuel can be defined in one row but it is also possible to divide them by additional categories such as efficiency etc.

		limit heat chp	capacity heat chp	capacity elec chp	limit hp	capacity hp	efficiency hp	efficiency heat chp	efficiency elec chp	fuel	source region
DE01	N1										DE01
	N3										DE
	N4										DE
DE02	N1										DE02
	N2										DE02
	N3										DE
	N4										DE
	N5										DE
…	…	…	…	…	…	…	…	…	…	…	…

INDEX

level 0: `str` Region (e.g. DE01, DE02).

level 1: `str` Name, arbitrary.

COLUMNS

limit heat chp: `float`, [MWh] The absolute maximum limit of heat produced by chp within the whole modeling period.

capacity heat chp: `float`, [MW] The installed heat capacity of all chp plants of the same group in the region.

capacity elect chp: `float`, [MW] The installed electricity capacity of all chp plants of the same group in the region.

limit hp: `float`, [MWh] The absolute maximum limit of heat produced by the heat plant within the whole modeling period.

capacity hp: `float`, [MW] The installed heat capacity of all heat of the same group in the region.

efficiency hp: `float`, [-] The average overall efficiency of the heat plant.

efficiency heat chp: `float`, [-] The average overall heat efficiency of the chp.

efficiency elect chp: `float`, [-] The average overall electricity efficiency of the chp.

fuel: `str`, [-] The used fuel of the plants. The fuel name must be equal to the fuel type of the commodity sources. The combination of *fuel* and *source region* has to exist in the *commodity sources* table.

source_region, [-] The source region of the fuel source. Typically this is the region of the index or `DE` if it is a global commodity source.

3.3.5 Mobility sector (optional)

Mobility demand series

`key:` 'mobility series', `value:` pandas.DataFrame()

The mobility demand can be entered regionally or supra-regional. However, it is recommended to define the mobility demand supra-regional except for *electricity*. The demand for electric mobility has be defined regional because it will be connected to the electricity bus of each region. The combination of region and name has to exist in the *mobility* table.

	DE01	DE02	…	DE
	electricity	electricity		N1
Time step 1				
Time step 2				
…	…	…	…	…

INDEX

time step: `int` Number of time step. Must be uniform in all series tables.

COLUMNS

unit: `[MW]`

level 0: `str` Region (e.g. DE01, DE02 or DE).

level 1: `str` Specification of the series e.g. "electricity" for each region or "diesel", "petrol" for DE.

Mobility

`key`: 'mobility', `value`: pandas.DataFrame()

This sheet covers the technologies of the mobility sector.

		efficiency	source	source region
DE01	electricity		electricity	DE01
DE02	electricity		electricity	DE02
...				
DE	N1		oil/biofuel/H2/etc	DE

INDEX

level 0: `str` Region (e.g. DE01, DE02 or DE).

level 1: `str` Name, arbitrary.

COLUMNS

efficiency: `float`, [-] The efficiency of the fuel production. If a *diesel* demand is defined in the *mobility demand series* table the *efficiency* represents the efficiency of *diesel* production from the commodity source e.g. oil. For a *biofuel* demand the efficiency of the production of *biofuel* from *biomass* has to be defined.

source: `str`, [-] The source that the technology uses. Except for *electricity* the combination of *source* and *source region* has to exist in the *commodity sources* table. The *electricity* source will be connected to the electricity bus of the region defined in *source region*.

source region: `str`, [-] The region where the source comes from.

3.3.6 Other (optional)

Storages

`key`: 'storages', `value`: pandas.DataFrame()

Different type of storages can be defined in this table. All different storage technologies (pumped hydro, batteries, compressed air, hydrogen, etc) have to be entered in a general way. Each row can indicate one storage or a group of storages. If the storage medium is electricity, then the storage must exist in a region DEXX. Otherwise, the storage can be defined under DE. It is possible to add additional columns for information purposes.

		storage medium	energy content	energy inflow	charge capacity	discharge capacity	charge efficiency	discharge efficiency	loss rate
DE01	S1	electricity							
	S2	electricity							
DE02	S1	electricity							
DE	S3	hydrogen							
…	…	…	…	…	…	…	…	…	…

INDEX

level 0: `str` Region (e.g. DE01, DE02).

level 1: `str` Name, arbitrary.

COLUMNS

storage medium: `str` The medium used to store energy. The storage medium must be defined in commodities, or it must be electricity.

energy content: `float`, [MWh] The maximum energy content of a storage or a group storages.

energy inflow: `float`, [MWh] The amount of energy that will feed into the storage of the model period in MWh. For example a river into a pumped hydroelectric energy storage.

charge capacity: `float`, [MW] Maximum capacity to charge the storage or the group of storages.

discharge capacity: `float`, [MW] Maximum capacity to discharge the storage or the group of storages.

charge efficiency: `float`, [-] Charging efficiency of the storage or the group of storages.

discharge efficiency: `float`, [-] Discharging efficiency of the storage or the group of storages.

loss rate: `float`, [-] The relative loss of the energy content of the storage. For example a loss rate or *0.01* means that the energy content of the storage will be reduced by *1%* in each time step.

Other converters

`key:` 'other converters', `value:` pandas.DataFrame()

Here, other converters than the ones already set, can be defined for linking different buses. A good example here is an electrolyser which connects electricity with hydrogen. Each converter has a source and a target bus with their respective regions. Other converter´s format is analogous to that of power plants and heat plants.

		capacity	annual limit	efficiency	variable costs	downtime factor	source	source region	target	target region
DE	electrolyser1						electricity	DE01	hydrogen	DE
DE	electrolyser2						electricity	DE02	hydrogen	DE
DE01	C1						S1	DE01	T1	DE01

INDEX

level 0: `str` Region (e.g. DE01, DE02).

level 1: `str` Name, arbitrary. The combination of region and name is the unique identifier for the converter or the group of converters.

COLUMNS

capacity: `float`, [MW] The installed capacity of the converter or the group of converters.

annual limit: `float`, [MWh] The absolute maximum limit of produced target units within the whole modeling period.

efficiency: `float`, [-] The average overall efficiency of the converter or the group of converters.

variable_costs: `float`, [€/MWh] The variable costs per produced target unit.

downtime_factor: `float`, [-] The time fraction of the modeling period in which the converter or the group of converters cannot produce target units. The installed capacity will be reduced by this factor `capacity * (1 - downtime_factor)`.

source: `str`, [-] The source bus of the converter or group of converters. The combination of *source_region* and *source* must exist in the commodity sources table or it can be electricity with its region DEXX.

source_region, [-] The source region of the source. Typically this is the region of the index or `DE` if it is a global commodity source.

target: `str`, [-] The target bus of the converter or group of converters. The combination of *target_region* and *target* must exist in the commodity sources table or it can be electricity with its region DEXX.

trget_region, [-] The target region of the target. Typically this is the region of the index or `DE` if it is a global commodity target.

Other demand series

`key:` 'other demand series', `value:` pandas.DataFrame()

Here, other demands different from electricity, heat or mobility can be entered as time series. Examples are hydrogen or synthetic fuel for the industry sector. The demands can be entered regionally under DEXX or supra-regional under DE. The format here is analogous to that of electricity, heat and mobility demand series.

	DE01		DE02		DE	
	D1	D2	D1	D3	hydrogen	syn fuel
	sector 1	sector 1	sector 2	sector 3	industry	industry
Time step 1						
Time step 2						
…	…	…	…	…	…	…

INDEX

time step: `int` Number of time step. Must be uniform in all series tables.

COLUMNS

unit: `[MW]`

level 0: `str` Region (e.g. DE01, DE02 or DE).

level 1: `str` Name. Specification of the series e.g. *hydrogen*, *syn fuel*.

level 2: `str` Sector name. Specification of the series e.g. *industry*, *LULUCF*. This extra level is used to differentiate the sector in which the commodity is used, since the same commodity may be used in different sectors.

Demand response

`key`: ‘demand response’, `value`: pandas.DataFrame()

Demand response, also known as demand side management is used to represent flexibility in the demand time series. Because of that it is applied on the four different demand series. There is the option of using two different methods of demand response: the interval and the delay one. The documentation of both methods con be found in SinkDSM where the interval method corresponds to “oemof” and the delay to “DIW” method. Depending on whether the interval or delay method is used, the shift interval or delay columns must be used. Finally, there is also the option of adding a price to use this feature.

				capacity up	capacity down	method	shift interval	delay	cost up	cost down
mobility demand series	DE01	electricity	None			interval	8	0		
	DE02	electricity	None			interval	8	0		
	DE	oil	None			delay	0	10		
electricity demand series	DE01	all	None			interval	8	0		
	DE02	indsutry	None			interval	8	0		
	DE02	buildings	None			interval	8	0		
heat demand series	DE01	heat pump	None			interval	6	0		
	DE	natural gas	None			delay	6	0		
other demand series	DE	hydrogen	indsutry			delay	0	12		

INDEX

level 0: `str` Name of the demand serie.

level 1: `str` Region (e.g. DE01, DE02 or DE)

level 2: `str` Specification of the serie. The combination of *region* and *specification of the serie* has to exist in the corresponding *demand serie* sheet.

level 3: `str` Sector name. This extra index is for when *other demand series* is used. If this is not the case, just write *None* instead.

COLUMNS

capacity up: `float`, [MW] The maximum limit with respect to the demand, to which the demand can be increased.

capacity down: `float`, [MW] The minimum limit with respect to the demand, to which the demand can be reduced.

method: `str`, [-] The method chosen to be used.

shift interval: `str`, [-] If the interval method is used, this column indicates the maximum interval that the demand can be shifted.

delay: `str`, [-] If the deelay method is used, this column indicates the maximum delay that demand can be shifted.

cost up: `float`, [€/MWh] The variable costs per shifted up unit

cost down: `float`, [€/MWh] The variable costs per shifted down unit.

3.4 Results

All results are stored in the *`results`* attribute of the *`DeflexScenario`* class. It is a dictionary with the following keys:

- main – Results of all variables (result dictionary from oemof.solph)
- param – Input parameter
- meta – Meta information and tags of the scenario
- problem – Information about the linear problem such as *lower bound*, *upper bound* etc.
- solver – Solver results
- solution – Information about the found solution and the objective value

The `deflex` package provides some analyse functions as described below but it is also possible to write your own post processing based on the oemof.solph API. See the results chapter of the oemof.solph documentation to learn how to handle the results.

3.4.1 Restore results

Most postprocessing functions need the results dictionary of the `DeflexScenario` as an input. So it is possible to restore only the results dictionary. Nevertheless, also the whole `DeflexScenario` object can be restored.

- *`restore_scenario()`* – restore a full scenario
- *`restore_results()`* – restore only the results dictionary.

Both function need the full file name (including the path) to the dumped scenario as input parameter. If you have many dumped files onn your hard disc you can use a search function to find and filter the files.

- *`search_dumped_scenarios()`* – search dump files on your hard disc.

The output of the search function can be directly used in the restore functions from above.

3.4.2 Postprocessing

There are different types of postprocessing functions available. Some can be used to verify the overall behaviour of the model. This can be used for debugging but also for plausibility checks. Some can be used to calculated additional key values from the results or to prepare the results to calculate further values. Furthermore, it is possible to get the result from all model variables in the `xlsx` or `csv` format.

For most postprocessing calculations cycles can cause problems because assumptions are needed on how to deal with the cycles and it is difficult to implement all possible assumptions in the functions. Therefore it might be easier to use the basic preparation functions and write your own calculations. See below on how to identify different kind of cycles.

3.4.3 Custom postprocessing

For a custom post processing it is possible to filter, group and prepare the results to ones own needs. Use dictionary and list comprehensions to find the needed flows and groups. The label and the class of the nodes can be used to filter the nodes.

The keys of the `results["main"]` dictionary are tuples.

- FLows: (<from_node>, <to_node>)
- Components (<component>, None)
- Buses (<bus>, None)

A node can be a component or a bus. The values of the tuples are the objects or None.

Get the keys of all buses:

```
from oemof.solph import Bus
bus_keys = [k for k in results["Main"].keys()
            if isinstance(k[0], Bus) and k[1] is None]
```

Get a list of buses:

```
from oemof.solph import Bus
buses = [k[0] for k in results["Main"].keys()
         if isinstance(k[0], Bus) and k[1] is None]
```

Get a table of all flows from *pv* sources:

Long version:

```
import pandas as pd
pv_keys = [
    k
    for k in results["Main"].keys()
    if k[0].label.tag == "volatile" and k[0].label.subtag == "solar"
]
pv = {}
for pv_key in pv_keys:
    pv[dflx.label2str(pv_key[0].label)] = results["Main"][pv_key][
        "sequences"
    ]["flow"]
print(pd.DataFrame(pv))
```

Short version:

```
import pandas as pd
pv = {
    dflx.label2str(k[0].label): v["sequences"]["flow"]
    for k, v in results["Main"].items()
    if k[0].label.tag == "volatile" and k[0].label.subtag == "solar"
}
print(pd.DataFrame(pv))
```

For more information about the results handling also see the results chapter of the oemof.solph documentation.

The following table gives an overview over the used classes and the naming of the label of the deflex components and buses. Each label is a nametuple with the fields *cat*, *tag*, *subtag* and *region*.

Table 1: Classes and labels of deflex nodes

	class	cat	tag	subtag	region
commodity bus	Bus	commodity	all	<fuel>	<region>
electricity bus	Bus	electricity	all	all	<region>
district heating bus	Bus	heat	district	all	<region>
decentralised heat bus	Bus	heat	decen-tralised	<fuel>	<region>
mobility bus	Bus	mobility	all	<name>	<region>
shortage source	Source	shortage	<cat of bus>	<subtag of bus>	<region>
commodity source	Source	source	commodity	<fuel>	<region>
volatile source	Source	source	volatile	<name>	<region>
power line	Transformer	line	electricity	<from region>	<to region>
mobility system	Transformer	mobility system	<name>	<fuel>	<region>
chp plant	Transformer	chp plant	<name>	<fuel>	<region>
decentralised heat system	Transformer	decentralised heat	<name>	<fuel>	<region>
heat plant	Transformer	heat plant	<name>	<fuel>	<region>
power plant	Transformer	power plant	<name>	<fuel>	<region>
other converter	Transformer	other converter	<name>	<fuel>	<region>
excess sink	Sink	excess	<cat of bus>	<subtag of bus>	<region>
electricity demand	Sink	electricity demand	electricity	<name>	<region>
district heat demand	Sink	heat demand	district	all	<region>
decentralised heat demand	Sink	heat demand	decen-tralised	<fuel>	<region>
mobility demand	Sink	mobility demand	mobility	<name>	<region>
other demand	Sink	other demand	other	<fuel>	<region>
storages	GenericStor-age	storage	<medium>	<name>	<region>

3.4.4 Export all results

To export the results from all variables into the `xlsx` or `csv` format, the results can be stored in a collection of pandas.DataFrame. This collection can be stored into a file. An example for this workflow can be found in the documentation of the function:

- *`get_all_results()`* – get all results as dictionary
- *`dict2file()`* – store the dictionary into a file

3.4.5 Get common values from results

The following values will be returned on an hourly base:

- marginal costs [EUR/MWh]
- highest emission [tons/MWh]
- lowest emission [tons/MWh]

- marginal costs power plant [-]
- emission of marginal costs power plant [tons/MWh]
- `deflex.calculate_key_values()` – get key values on an hourly base

At the moment this works only with hourly time steps. This function is still work in progress and may return more key values in the future. Please write an issue on github for a discussion about further values.

3.4.6 Analyse flow cycles

As a directed graph is used to define an energy system. Cycles are defined as a group of successive directed flows, where the first and the last node or bus are the same. Small cycles are all storages. As this is a trivial solution of a cycle analysis storages can be excluded. Another kind of cycles are the combination of electrolysis and hydrogen power plants. Power lines will also cause cycles. Pure power line cycles can also be excluded but this will not exclude a cycle cause by an electrolysis in one region and a hydrogen power plant in another even though a power line is included in this cycle.

A cycle may not be a problem if it is not used as a cycle in the system. So it is also possible to analyse the usage of the cycle:

1. cycle – a cycle that can be used within the model
2. used cycle – a cycle in which all involved flows are used at least once.
3. suspicious cycle – a cycle in which all involved flows are used within one time step.

The following functions are available

- `Cycles()` – initialise a Cycle object
- `cycles()` – all cycles in one table per cycle
- `used_cycles()` – all used cycles in one table per cycle
- `suspicious_cycles()` – all suspicious cycles in one table per cycle
- `get_suspicious_time_steps()` – get the time steps in which all flows are active
- `print()` – print an overview of all existing cycles
- `details()` – print a more detailed overview of all existing cycles

3.4.7 Analyse the energy system graph

It is possible to convert the graph of the EnergySystem class into an nxgraph of networkx. So, it is possible to use all methods and functions of networkx associate with a directed graph (DiGraph). Furthermore, deflex provides some function to associate colors with types of nodes or with the total weight of an edge (flow). This can be used if the graph is exported to a `graphml` file. Such a file can be opened in e.g. yEd where the colors can be used to display the nodes and edges in the associated colors.

- `DeflexGraph()` – initialise a *DeflexGraph* object
- `nxgraph()` – get an *DiGraph* of networkx
- `write()` – export the graph to a *graphml* file
- `color_edges_by_weight()` – associate a color from a color map according to the total weight
- `color_nodes_by_type()` – associate a color by the type of the node
- `color_nodes_by_substring()` – associate a color by a substring of the label of the node

- `group_nodes_by_type()` – group all nodes of the graph by their type

3.4.8 Get dual variables

The dual variable is available for all buses in the energy system.

`fetch_dual_results()` – Get the resulta of the dual variables of all buses in one table

3.4.9 CHP allocation

These tool are mostly not connected to deflex but could be used in any context. The functions just implement typical allocation methods in Python code:

- `allocate_fuel_deflex()` – allocate the fuel with default values from a config file
- `allocate_fuel()` – allocate the fuel with all values defined by the user
- `efficiency_method()` – efficiency method
- `exergy_method()` – carnot or exergy method
- `finnish_method()` – alternative_generation or finnish method
- `iea_method()` – IEA method

3.4.10 Arrange parts of the results

This parts can be used for plots and identification of the model

- `solver_results2series()` – get the results returned from the external solver
- `meta_results2series()` – get some general and meta results
- `group_buses()` – group all buses by label
- `get_time_index()` – get the used time index
- `nodes2table()` – get an overview about all nodes and their total in- and outflows

3.4.11 Combine results and parameter

The following functions can be used for further calculations. See the examples for more information.

- `fetch_converter_parameters()` – get all values related to the converter
- `fetch_attributes_of_commodity_sources()` – get the values of the commodity sources
- `get_combined_bus_balance()` – combine buses in a multiregion model
- `get_converter_balance()` – the energy balance around converter to calculate emissions and costs

TABLE of LABELS!!!!

3.5 Plots

Deflex does not include plotting function as plotting is mostly a very individual part and there are already a lot of useful packages available. Nevertheless, deflex provides maps for the default region sets and some example on how to create spatial plots. The maps can be access using the following functions.

- *`deflex_geo()`* – Get the default maps of deflex
- *`divide_off_and_onshore()`* – distinguish offshore and onshore regions in a given map

3.6 General tools

Solph and deflex use logging messages to give a feedback from the running program, so deflex provides an easy function to activate the logger on the INFO level:

- *`use_logging()`*

Some functions does not return a table but a set of table. To store these set of tables in a xlsx-map or a collection of csv-files the following function can be used.

- *`dict2file()`*

CHAPTER 4

Reference

4.1 Scenario

4.1.1 Scenario class

`deflex.DeflexScenario`([meta, input_data, . . .])	The Deflex Scenario is the center of a deflex energy model.

deflex.DeflexScenario

class deflex.**DeflexScenario**(*meta=None*, *input_data=None*, *es=None*, *results=None*)

The Deflex Scenario is the center of a deflex energy model. It can store the needed input data and the results after a successful optimisation. a inherits from the Scenario class and extends the Scenario class with valid nodes creation. Additionally one can define an extra_regions attribute to create an extra commodity source for these regions. This makes it possible to create a source balance for these regions.

Parameters

- **meta** (*dict*) – Meta information of the DeflexScenario (optional).
- **input_data** (*dict*) – A dictionary of tables in the deflex scenario style (optional).
- **es** (*oemof.solph.EnergySystem*) – An Energy system (optional).
- **results** (*dict*) – A valid Deflex results dictionary (optional).

input_data

The input data is organised in a dictionary of pandas.DataFrame/ pandas.Series. The keys are the data names (string) and the values are the data tables.

Type dict

results

There are different sub-sections of the results. The dictionary has got the following keys:

- main – Results of all variables (result dictionary from oemof.solph)
- param – Input parameter
- meta – Meta information and tags of the scenario
- problem – Information about the linear problem such as *lower bound*, *upper bound* etc.
- solver – Solver results
- solution – Information about the found solution and the objective value

The model results are stored in the *main* section. It contains another dictionary with tuples as keys and the results of the variables as values (nested dictionary with pandas.DataFrame). The tuples contain the node object in the following form: (from_node, to_node) for flows and (node, None) for components. See the solph documentation for more details.

Type dict

meta

Meta information that can be used to search for in stored scenarios. The dictionary keys can be used like tags or categories.

Type dict

es

This attribute will hold the oemof.solph.EnergySystem.

Type oemof.solph.EnergySystem

__init__(*meta=None*, *input_data=None*, *es=None*, *results=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([meta, input_data, es, results])	Initialize self.
add_nodes_to_es(nodes)	Add nodes to an existing solph.EnergySystem.
check_input_data()	Check the input data for NaN values.
compute([solver, with_duals])	Create a solph.Model from the input data and optimise it using an external solver.
create_model()	Create a solph model from an EnergySystem object.
create_nodes()	Creates solph components and buses from the input data and store them in a dictionary with unique IDs as keys.
dump(filename)	Store a solved scenario class into the binary pickle format.
initialise_energy_system()	Create a solph.EnergySystem and store it in the es attribute.
read_csv(path)	Load scenario from a csv-collection.
read_xlsx(filename)	Load scenario data from an xlsx file.
solve(model[, solver, with_duals])	Solve the solph.Model.
store_graph(filename, **kwargs)	Store the EnergySystem graph into a *.graphml* file.
table2es()	Create a populated solph.EnergySystem from the input data.
to_csv(path)	Store the input data as a csv-collection.
to_xlsx(filename)	Store the input data into an xlsx-file.

4.1.2 Read/Write a scenario

deflex.DeflexScenario.read_xlsx(filename)	Load scenario data from an xlsx file.
deflex.DeflexScenario.read_csv(path)	Load scenario from a csv-collection.
deflex.create_scenario(path[, file_type])	Create a deflex scenario object from file.
deflex.search_input_scenarios(path[, csv, ...])	Search for files with an .xlsx extension or directories ending with '_csv'.
deflex.DeflexScenario.to_xlsx(filename)	Store the input data into an xlsx-file.
deflex.DeflexScenario.to_csv(path)	Store the input data as a csv-collection.
deflex.DeflexScenario.dump(filename)	Store a solved scenario class into the binary pickle format.
deflex.DeflexScenario.store_graph(filename, ...)	Store the EnergySystem graph into a *.graphml* file.

deflex.DeflexScenario.read_xlsx

DeflexScenario.**read_xlsx**(*filename*)

Load scenario data from an xlsx file. The full path has to be passed.

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat.xlsx")
>>> sc = dflx.DeflexScenario()
>>> len(sc.input_data)
0
>>> sc = sc.read_xlsx(fn)
>>> len(sc.input_data)
11
```

deflex.DeflexScenario.read_csv

DeflexScenario.**read_csv**(*path*)

Load scenario from a csv-collection. The path of the directory has to be passed.

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.DeflexScenario()
>>> len(sc.input_data)
0
>>> sc = sc.read_csv(fn)
>>> len(sc.input_data)
11
```

deflex.create_scenario

`deflex.`**`create_scenario`**(*path*, *file_type=None*)

Create a deflex scenario object from file.

Parameters

- **`path`** (*`str`*) – A valid deflex scenario file.
- **`file_type`** (*`str or None`*) – Type of the input data. Valid values are 'csv', 'xlsx', None. If the input is non the path should end on 'csv', '.xlsx' to allow auto-detection.

Returns

Return type *deflex.DeflexScenario*

Examples

```
>>> from deflex import fetch_test_files, TEST_PATH
>>> fn = fetch_test_files("de17_heat.xlsx")
>>> s = create_scenario(fn, file_type="xlsx")
>>> type(s)
<class 'deflex.scenario.DeflexScenario'>
>>> int(s.input_data["volatile plants"]["capacity"]["DE01", "wind"])
3815
>>> type(create_scenario(fn))
<class 'deflex.scenario.DeflexScenario'>
>>> create_scenario(fn, file_type="csv"
...     )  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
 ...
NotADirectoryError: [Errno 20] Not a directory:
```

deflex.search_input_scenarios

`deflex.`**`search_input_scenarios`**(*path*, *csv=True*, *xlsx=False*, *exclude=None*)

Search for files with an .xlsx extension or directories ending with '_csv'.

By now it is not possible to distinguish between valid deflex scenarios and other xlsx-files or directories ending with '_csv'. Therefore, the given directory should only contain valid scenarios.

The function will not search recursively.

Parameters

- **`path`** (*`str`*) – Directory with valid deflex scenarios.
- **`csv`** (*`bool`*) – Search for csv directories.
- **`xlsx`** (*`bool`*) – Search for xls files.
- **`exclude`** (*`str`*) – A string that is not allowed in the filename. Filenames containing this strings will be excluded.

Returns **Scenarios found in the given directory**

Return type list

Examples

```
>>> import shutil
>>> from deflex import fetch_test_files, search_input_scenarios
>>> test_file = fetch_test_files("de02_heat.xlsx")
>>> test_path = os.path.dirname(test_file)
>>> my_csv = search_input_scenarios(test_path)
>>> len(my_csv)
16
>>> os.path.basename(my_csv[0])
'de02_heat_csv'
>>> my_xlsx = search_input_scenarios(test_path, csv=False, xlsx=True)
>>> len(my_xlsx)
17
>>> os.path.basename([e for e in my_xlsx][0])
'de02_heat.xlsx'
>>> len(search_input_scenarios(test_path, xlsx=True))
33
>>> scenario = create_scenario([e for e in my_xlsx][0])
>>> csv_path = os.path.join(test_path, "de02_new_csv")
>>> scenario.to_csv(csv_path)
>>> len(search_input_scenarios(test_path, xlsx=True))
34
>>> len(search_input_scenarios(test_path, xlsx=True, exclude="de02"))
25
>>> len(search_input_scenarios(test_path, xlsx=True, exclude="test"))
34
>>> shutil.rmtree(csv_path)  # remove test results, skip this line to go on
```

deflex.DeflexScenario.to_xlsx

`DeflexScenario.`**`to_xlsx`**`(`*filename*`)`

Store the input data into an xlsx-file.

filename [str] Full path to the filename.

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.DeflexScenario()
>>> # read scenario from xlsx-file
>>> sc = sc.read_csv(fn)
>>> # store scenario as csv-collection.
>>> sc.to_xlsx(fn.replace("_csv", ".xlsx"))
```

deflex.DeflexScenario.to_csv

`DeflexScenario.`**`to_csv`**`(`*path*`)`

Store the input data as a csv-collection.

filename [str] Full path to the filename.

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat.xlsx")
>>> sc = dflx.DeflexScenario()
>>> # read scenario from xlsx-file
>>> sc = sc.read_xlsx(fn)
>>> # store scenario as csv collection.
>>> sc.to_csv(fn.replace(".xlsx", "_csv"))
```

deflex.DeflexScenario.dump

DeflexScenario.**dump**(*filename*)

Store a solved scenario class into the binary pickle format.

The file will be stored with the suffix *.dflx*. If the given filename does not contain the suffix, it will be added to the filename.

It is possible to restore the dump but it is not possible to compute a restored dump. Unsolved scenarios should be stored in the xlsx or csv format.

```
>>> import os
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.create_scenario(fn, "csv")
>>> sc.results is None
True
>>> sc.compute()  # doctest: +ELLIPSIS
Welcome to the CBC MILP ...
>>> fn_dump = fn.replace("_csv", ".dflx")
>>> os.path.basename(fn_dump)
'de02_no-heat.dflx'
>>> sc.dump(fn_dump)
>>> os.path.isfile(fn_dump)
True
>>> sc2 = dflx.restore_scenario(fn_dump)
>>> type(sc2)
<class 'deflex.scenario.DeflexScenario'>
>>> sc2.results.keys()
['Problem', 'Solver', 'Solution', 'Main', 'Param', 'Meta']
>>> os.remove(fn_dump)
```

deflex.DeflexScenario.store_graph

DeflexScenario.**store_graph**(*filename*, ***kwargs*)

Store the EnergySystem graph into a *.graphml* file.

The kwargs are passed to the oemof.network function create_nx_graph().

Parameters **filename** (*str*) – Full path of the graphml-file.

Examples

```
>>> import os
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.create_scenario(fn, "csv")
>>> sc.table2es()
>>> fn_graph = fn.replace("_csv", ".graphml")
>>> os.path.basename(fn_graph)
'de02_no-heat.graphml'
>>> sc.store_graph(fn_graph)
>>> os.path.isfile(fn_graph)
True
>>> os.remove(fn_graph)
```

4.1.3 Compute scenario

deflex.DeflexScenario.compute([solver, ...])	Create a solph.Model from the input data and optimise it using an external solver.

deflex.DeflexScenario.compute

DeflexScenario.**compute**(*solver='cbc'*, *with_duals=True*, ***kwargs*)

Create a solph.Model from the input data and optimise it using an external solver. Afterwards the results are stored in the results attribute.

Parameters

- **solver** (*str*) – The name of the solver as used in the Pyomo package like cbc, glpk, gurobi, cplex... (default: cbc).
- **with_duals** (*bool*) – Receive the dual variables of all buses in the results (default: True).

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.create_scenario(fn, "csv")
>>> sc.results is None
True
>>> sc.compute()  # doctest: +ELLIPSIS
Welcome to the CBC MILP ...
>>> sc.results.keys()
['Problem', 'Solver', 'Solution', 'Main', 'Param', 'Meta']
```

4.1.4 Advanced scenario methods

deflex.DeflexScenario.check_input_data()	Check the input data for NaN values.
deflex.DeflexScenario.table2es()	Create a populated solph.EnergySystem from the input data.
deflex.DeflexScenario.create_model()	Create a solph model from an EnergySystem object.

Continued on next page

Table 5 – continued from previous page

deflex.DeflexScenario.create_nodes()	Creates solph components and buses from the input data and store them in a dictionary with unique IDs as keys.
deflex.DeflexScenario.solve(model[, solver, ...])	Solve the solph.Model.
deflex.DeflexScenario.initialise_energy_system()	Create a solph.EnergySystem and store it in the es attribute.
deflex.DeflexScenario.add_nodes_to_es(nodes)	Add nodes to an existing solph.EnergySystem.

deflex.DeflexScenario.check_input_data

DeflexScenario.**check_input_data**()

Check the input data for NaN values. If warning is True (default: False) a warning for all tables is raised that contain NaN values. This is useful if you suspect many NaN values in your data set, so you get a good overview over all the corrupt columns. Otherwise an exception is raised on the first occurrence of NaN values.

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.create_scenario(fn, "csv")
>>> sc.input_data["electricity demand series"].iloc[15] = float("nan")
>>> sc.input_data["volatile series"].iloc[11] = float("nan")
>>> sc.check_input_data()  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
...
ValueError: NaN values found in the following tables: electricity...
```

deflex.DeflexScenario.table2es

DeflexScenario.**table2es**()

Create a populated solph.EnergySystem from the input data.

The EnergySystem object will be stored in the `es` attribute of the *DeflexScenario*.

This method is included in the *compute()* method and is only needed for advanced usage.

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.create_scenario(fn, "csv")
>>> sc.es is None
True
>>> sc.table2es()
>>> type(sc.es)
<class 'oemof.solph.network.energy_system.EnergySystem'>
```

deflex.DeflexScenario.create_model

`DeflexScenario.`**`create_model`**`()`

Create a solph model from an EnergySystem object.

This method is included in the `compute()` method and is only needed for advanced usage.

Returns

Return type oemof.solph.Model

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.create_scenario(fn, "csv")
>>> sc.table2es()
>>> type(sc.create_model())
<class 'oemof.solph.models.Model'>
```

deflex.DeflexScenario.create_nodes

`DeflexScenario.`**`create_nodes`**`()`

Creates solph components and buses from the input data and store them in a dictionary with unique IDs as keys.

Returns

Return type dict

deflex.DeflexScenario.solve

`DeflexScenario.`**`solve`**`(`*model*, *solver='cbc'*, *with_duals=True*, ***solver_kwargs*`)`

Solve the solph.Model. This method is included in the `compute()` method and is only needed for advanced usage.

Parameters

- **`model`** (*`oemof.solph.Model`*) –
- **`solver`** (*`str`*) –
- **`with_duals`** (*`bool`*) –

Other Parameters

- **tee** (*bool*) – Set to *False* to suppress the solver output (default: True).
- **logfile** (*str*) – Define the path where to store the log file of the solver.

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat_csv")
>>> sc = dflx.create_scenario(fn, "csv")
>>> sc.table2es()
>>> my_model = sc.create_model()
```

(continues on next page)

(continued from previous page)

```
>>> sc.solve(my_model, with_duals=False)  # doctest: +ELLIPSIS
Welcome to the CBC MILP ...
>>> sc.results.keys()
['Problem', 'Solver', 'Solution', 'Main', 'Param', 'Meta']
```

deflex.DeflexScenario.initialise_energy_system

`DeflexScenario.`**`initialise_energy_system`**`()`

Create a solph.EnergySystem and store it in the es attribute. The input_data attribute has to contain the input data to use this method.

Returns

Return type self

deflex.DeflexScenario.add_nodes_to_es

`DeflexScenario.`**`add_nodes_to_es`**`(`*nodes*`)`

Add nodes to an existing solph.EnergySystem. If the EnergySystem does not exist an Error is raised. This method is included in the `compute()` method and is only needed for advanced usage.

Parameters **`nodes`** (*dict*) – Dictionary with a unique key and values of type oemof.network.Node.

Returns

Return type self

4.2 Scripts

4.2.1 Python scripts

`deflex.model_scenario`([path, file_type, . . .])	Compute a deflex scenario with the full work flow:
`deflex.batch_model_scenario`(path[, . . .])	Model a single scenario in batch mode.
`deflex.model_multi_scenarios`(scenarios[, . . .])	Model multi scenarios in parallel.

deflex.model_scenario

`deflex.`**`model_scenario`**`(`*path=None*, *file_type=None*, *dump=True*, *results=False*, *solver='cbc'*`)`

Compute a deflex scenario with the full work flow:

- creating a scenario
- loading the input data
- computing the scenario
- storing the results

Parameters

- **path** (*str or None*) – File or directory with a valid deflex scenario. If no path is given an energy system (es) has to be passed.
- **file_type** (*str or None*) – Type of the input data. Valid values are 'csv', 'xlsx', None. If the input is non the path should end on 'csv' or '.xlsx'.
- **dump** (*str or bool*) – Path to store the dump file. If True the results will be stored along with the scenarios using the same name and the suffix *.dflx*. If False no dump will be stored (default: True).
- **results** (*str or bool*) – Path to store the results in an spreadsheet. If True the results will be stored along with the scenarios using the same name and the suffix *_results.xlsx*. If False no results will be stored (default: False).
- **solver** (*str*) – The solver to use for the optimisation (default: cbc).

Examples

```
>>> from deflex import fetch_test_files, TEST_PATH
>>> fn = fetch_test_files("de02_no-heat.xlsx")
>>> r = model_scenario(fn, file_type="xlsx", dump=True
...     )  # doctest: +ELLIPSIS
Welcome to the CBC MILP ...
>>> os.remove(fn.replace(".xlsx", ".dflx"))
```

deflex.batch_model_scenario

deflex.**batch_model_scenario**(*path*, *file_type=None*, *ignore_errors=True*, *flat_tuple=False*, ***kwargs*)

Model a single scenario in batch mode. By default errors will be ignored and returned together with the traceback.

Parameters

- **path** (*str*) – A valid deflex scenario.
- **file_type** (*str or None*) – Type of the input data. Valid values are 'csv', 'xlsx', None. If the input is non the path should end on 'csv', '.xlsx'.
- **ignore_errors** (*bool*) – Set True to stop the script if an error occurs for debugging. By default errors are ignored and returned.
- **flat_tuple** (*bool*) – Return a normal tuple instead of a named tuple. This is needed for multi-process use. (default: False)

Other Parameters

- **dump** (*str or bool*) – Path to store the dump file. If True the results will be stored along with the scenarios using the same name and the suffix *.dflx*. If False no dump will be stored (default: True).
- **results** (*str or bool*) – Path to store the results in an spreadsheet. If True the results will be stored along with the scenarios using the same name and the suffix *_results.xlsx*. If False no results will be stored (default: False).
- **solver** (*str*) – The solver to use for the optimisation (default: cbc).

Returns

Return type namedtuple

Examples

```
>>> from deflex import fetch_test_files
>>> fi = fetch_test_files("de02_heat_csv")
>>> r = batch_model_scenario(fi, ignore_errors=False)  # doctest: +ELLIPSIS
Welcome to the CBC MILP ...
>>> r.name
'de02_heat_csv'
>>> my_dump_file = r.dump
>>> os.path.basename(my_dump_file)
'de02_heat_csv.dflx'
>>> r.trace
>>> r.return_value.year > 2019
True
>>> f_wrong = os.path.join("wrong_file.xlsx")
>>> r = batch_model_scenario(f_wrong)
>>> r.name
'wrong_file.xlsx'
>>> repr(r.return_value)
"FileNotFoundError(2, 'No such file or directory')"
>>> r.results
>>> r.trace  # doctest: +ELLIPSIS
'Traceback (most recent call last):...
>>> os.remove(my_dump_file)
```

deflex.model_multi_scenarios

deflex.**model_multi_scenarios**(*scenarios*, *cpu_fraction=0.2*, *log_file=None*, *results=False*)

Model multi scenarios in parallel. Keep in mind that the memory usage is the critical resource for large models. So start with a low cpu_fraction to avoid memory errors.

Parameters

- **scenarios** (*iterable*) – Multiple scenarios to be modelled in parallel.
- **cpu_fraction** (*float*) – Fraction of available cpu cores to use for the parallel modelling. A resulting dezimal number of cores will be rounded up to an integer.
- **log_file** (*str*) – Filename to store the log file.
- **results** (*bool*) – Store an spreadsheet results file (default: False).

Examples

```
>>> from deflex import fetch_test_files, TEST_PATH
>>> fn1 = fetch_test_files("de03_fictive_csv")
>>> fn2 = fetch_test_files("de03_fictive_broken.xlsx")
>>> my_log_file = os.path.join(TEST_PATH, "my_log_file.csv")
>>> my_scenarios = [fn1, fn2]
>>> model_multi_scenarios(my_scenarios, log_file=my_log_file)
>>> my_log = pd.read_csv(my_log_file, index_col=[0])
>>> good = my_log.loc["de03_fictive_csv"]
>>> rv = good["return_value"]
>>> datetime.strptime(rv, "%Y-%m-%d %H:%M:%S.%f").year > 2019
True
>>> good["trace"]
```

(continues on next page)

(continued from previous page)

```
nan
>>> os.path.basename(good["dump"])
'de03_fictive_csv.dflx'
>>> good["results"]
False
>>> broken = my_log.loc["de03_fictive_broken.xlsx"]
>>> broken["return_value"].replace("'", "")  # doctest: +ELLIPSIS
'ValueError(Missing time series for geothermal (capacity: 12.56) in DE02...
>>> broken["trace"]  # doctest: +ELLIPSIS
'Traceback (most recent call last)...
>>> broken["dump"]
nan
>>> os.remove(my_log_file)
>>> os.remove(good["dump"])
```

4.2.2 Console scripts

deflex.console_scripts.main()	deflex-compute [-h] [–version] [–results [RESULTS]] [–dump [DUMP]] [–solver [SOLVER]] path
deflex.console_scripts.result()	deflex-results [-h] [–version] [–filetype [FILETYPE]] function in_path out_path

deflex.console_scripts.main

deflex.console_scripts.**main**()

deflex-compute [-h] [–version] [–results [RESULTS]] [–dump [DUMP]] [–solver [SOLVER]] path

Computing a deflex scenario. By default the name of the result file is derived from the name of the input file by adding '_results but it is possible to define a custom path. The results will be of the same file format as the input scenario.

Optionally a dump-file can be stored. If no path is given the path is derived from the path of the input scenario. The suffix of the dump is '.dflx'. The dump can be processed using *deflex_result*.

Positional Arguments

```
path      Input file or directory.
```

Optional Arguments

-h, --help	show this help message and exit
--version	show program's version number and exit
--results <RESULTS>	The name of the results file or directory or False to get no result file. By default the path is derived from scenario path.
--dump <DUMP>	The name of the dump file. Leave empty for the default file name
--solver <SOLVER>	Solver to use for computing (default cbc)

deflex.console_scripts.result

deflex.console_scripts.**result**()

deflex-results [-h] [–version] [–filetype [FILETYPE]] function in_path out_path

Processing the results from a computed deflex dump file. The following functions are available:

- calculate_key_values `calculate_key_values()`
- something

See the documentation for more details.

Positional Arguments

```
function                Post-processing function to use.
in_path                 Input file or directory.
out_path                Output file or directory.
```

Optional Arguments

-h, --help	show this help message and exit
--version	show program's version number and exit
--filetype <FILETYPE>	The file_type of the output file xlsx or csv. By default the suffix of the output file is used, if possible.

4.3 Postprocessing

4.3.1 Restore dumped scenarios

`deflex.search_dumped_scenarios`(path[, extension])	Filter results by extension and meta data.
`deflex.restore_scenario`(filename[, ...])	Restore a full Scenario from a dump file (*.dflx*).
`deflex.restore_results`(file_names[, ...])	Restore only the result dictionary from a dumped scenario or a list of dumped scenarios.

deflex.search_dumped_scenarios

`deflex.`**`search_dumped_scenarios`**(*path*, *extension='dflx'*, ***parameter_filter*)

Filter results by extension and meta data.

The function will search the $HOME folder recursively for files with the '.dflx' extension. Afterwards all files will filtered by the meta data.

If there is an *info* table in your input data, the keys and values can be used to filter the values. For example different region sets are defined as maps with *de21*, *de22* and *de17* and different years were modelled. Futhermore some are modelled with the heating sector (heat: True) and some not (heat: False). See the example below on how to search for these scenarios.

Parameters

- **`path`** (*`str`*) – Start folder from where to search recursively.
- **`extension`** (*`str`*) – Extension of the results files (default: ".dflx")
- **`**parameter_filter`** – Set filter always with lists e.g. map=["de21"] or map=["de21", "de22"]. The values in the list have to be strings. Two filters will be connected with 'AND', the values within one filter with *OR*. The filters year=["2014"], map=["de21", "de22"] will find all scenarios with: year==2014 and (map=="de21" or map=="de22")

Examples

```
>>> from deflex import TEST_PATH
>>> from deflex  import fetch_test_files
>>> my_file_name = fetch_test_files("de17_heat.dflx")
>>> res = search_dumped_scenarios(path=TEST_PATH, map=["de17"])
>>> len(res)
2
>>> sorted(res)[0].split(os.sep)[-1]
'de17_heat.dflx'
>>> res = search_dumped_scenarios(path=TEST_PATH, map=["de17", "de21"])
>>> len(res)
6
>>> res = search_dumped_scenarios(
...     path=TEST_PATH, map=["de17", "de21"], heat=["True"])
>>> len(res)
3
>>> sorted(res)[0].split(os.sep)[-1]
'de17_heat.dflx'
```

deflex.restore_scenario

`deflex.`**`restore_scenario`**(*filename*, *scenario_class=<class 'deflex.scenario.DeflexScenario'>*)

Restore a full Scenario from a dump file (*.dflx*).

If only the results are needed use `restore_results()` instead. By default a DeflexScenario is created but a different Scenario class can be passed The Scenario has to be equal to the dumped Scenario otherwise the restore will fail.

Parameters

- **filename** (*str*) – The path to the dumped file (*.dflx*).
- **scenario_class** (*class*) – A child of the deflex.Scenario class or the Scenario class itself.

Returns

Return type deflex.Scenario

deflex.restore_results

`deflex.`**`restore_results`**(*file_names*, *scenario_class=<class 'deflex.scenario.DeflexScenario'>*)

Restore only the result dictionary from a dumped scenario or a list of dumped scenarios. The results will be a deflex result dictionary with the following keys:

- main – Results of all variables
- param – Input parameter
- meta – Meta information and tags of the scenario
- problem – Information about the linear problem such as lower bound, upper bound etc.
- solver – Solver results
- solution – Information about the found solution and the objective value

Parameters

- **file_names** (*list or string*) – All file names (full path) that should be loaded.
- **scenario_class** (*class*) – A child of the deflex.Scenario class or the Scenario class itself.

Returns

- *A list of results dictionaries or a single dictionary if one file name is*
- **given** (*list or dict*)

Examples

```
>>> from deflex import fetch_test_files
>>> fn1 = fetch_test_files("de21_no-heat_transmission.dflx")
>>> fn2 = fetch_test_files("de02_no-heat.dflx")
>>> sorted(restore_results(fn1).keys())
['Input data', 'Main', 'Meta', 'Param', 'Problem', 'Solution', 'Solver']
>>> sorted(restore_results([fn1, fn2])[0].keys())
['Input data', 'Main', 'Meta', 'Param', 'Problem', 'Solution', 'Solver']
```

4.3.2 Analyse and draw graph

deflex.postprocessing.graph.Edge(**kwargs)	An edge of a DeflexGraph
deflex.DeflexGraph(results, **kwargs)	The deflex model graph with a networkx representation.
deflex.DeflexGraph.nxgraph(**kwargs)	Get a networkx.DiGraph() from the deflex results.
deflex.DeflexGraph.write(filename, **kwargs)	Write the graph into a .graphml file.
deflex.DeflexGraph.color_edges_by_weight([...])	Color all edges by their weight using a matplotlib color map (cmap).
deflex.DeflexGraph.color_nodes_by_type(colors)	Color all nodes in a specific color according to their class.
deflex.DeflexGraph.color_nodes_by_substring(colors)	Color all nodes in a specific color according to a given substring.
deflex.DeflexGraph.group_nodes_by_type([...])	Group all nodes by types returning a dictionary with the types or the name of the types as keys and the list of nodes as value.

deflex.postprocessing.graph.Edge

class deflex.postprocessing.graph.**Edge**(***kwargs*)

An edge of a DeflexGraph

All attributes can be defined using keyword arguments.

nodes

The node where the edge comes from and the node where it goes to.

Type tuple

label

A string representation of the *nodes* attribute.

Type str

sequence
A time series of the flow variable of the edge.

Type iterable

weight
The sum of the *sequence* attribute.

Type float

color
A color string for plots/drawings of the graph.

Type str

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(**kwargs)	Initialize self.

deflex.DeflexGraph

class deflex.**DeflexGraph**(*results*, ***kwargs*)
The deflex model graph with a networkx representation.

The sum of the variables are addedcan be used as weight

Parameters **results** (*dict*) – Deflex results dictionary.

Other Parameters

- **default_node_color** (*str*) – The default color as a dictionary with the keys "fg" for the foreground color (font color) and "bg" for the background color (fill color). The color has to be a hexadecimal string. The default color is used if no other color is set. (default: {"bg": "#6a6a72", "fg": "#000000"}")
- **default_edge_color** (*str*) – The default edge color as a hexadecimal string. The default color is used if no other color is set. (default: "#000000"")

nodes
All nodes of the deflex energy system graph.

Type list

edges
All edges of the deflex energy system graph.

Type list

default_node_color
The default color for nodes with the keys *bg* for the background color and *fg* for text color (foreground).

Type dict

default_edge_color
The default color for edges with the keys *bg* for the background color and *fg* for text color (foreground).

Type dict

Examples

```
>>> import os
>>> from deflex import fetch_test_files
>>> from deflex import restore_results
>>> from deflex import DeflexGraph
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> my_results = restore_results(fn)
>>> dflx_graph = DeflexGraph(my_results)
>>> len(dflx_graph.nodes)
226
>>> sorted(dflx_graph.nodes)[5].label
Label(cat='chp plant', tag='other', subtag='other', region='DE01')
>>> len(dflx_graph.edges)
323
>>> dflx_graph.edges[5].label
'chp-plant_lignite_lignite_DE01 -> heat_district_all_DE01'
>>> type(dflx_graph.edges[5])
<class 'deflex.postprocessing.graph.Edge'>
>>> dflx_graph.edges[5].weight
136524.0
>>> nx_graph = dflx_graph.nxgraph()
>>> nx.number_of_nodes(nx_graph)
226
>>> nx.number_weakly_connected_components(nx_graph)
1
```

__init__(*results*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(results, **kwargs)	Initialize self.
color_edges_by_weight([cmap, max_weight])	Color all edges by their weight using a matplotlib color map (cmap).
color_nodes_by_substring(colors)	Color all nodes in a specific color according to a given substring.
color_nodes_by_type(colors[, use_name])	Color all nodes in a specific color according to their class.
group_nodes_by_type([use_name])	Group all nodes by types returning a dictionary with the types or the name of the types as keys and the list of nodes as value.
nxgraph(**kwargs)	Get a networkx.DiGraph() from the deflex results.
write(filename, **kwargs)	Write the graph into a .graphml file.

deflex.DeflexGraph.nxgraph

DeflexGraph.**nxgraph**(***kwargs*)
Get a networkx.DiGraph() from the deflex results.

Some labels will be added to the edges and nodes.

Node * label: label of the Node as string * bg_color: background color of the node for plots or exports * fg_color: text color of the node for plots or exports * type: name of the node class

Edge * weight: sum of the flow variable * color: color of the edge depending of the weight

Other Parameters weight_exponent (*int*) – Shift the decimal point: $weight = weight \cdot 10^{weight\_exponent}$

Examples

```
>>> import os
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> dflx_graph = dflx.DeflexGraph(my_results)
>>> type(dflx_graph.nxgraph(weight_exponent=-3))
<class 'networkx.classes.digraph.DiGraph'>
>>> edges_data = dflx_graph.nxgraph().edges.data()
>>> wind2bus = [e[2] for e in edges_data if
...     e[0].subtag == "wind" and
...     e[0].region == "DE01"
... ][0]
>>> wind2bus["label"]
'source_volatile_wind_DE01 -> electricity_all_all_DE01'
>>> wind2bus["weigth"], wind2bus["color"]
('337.0', '#000000')
>>> dflx_graph.color_edges_by_weight()
>>> edges_data = dflx_graph.nxgraph(weight_exponent=-3).edges.data()
>>> wind2bus = [e[2] for e in edges_data if
...     e[0].subtag == "wind" and
...     e[0].region == "DE01"
... ][0]
>>> wind2bus["weigth"], wind2bus["color"]
('337.0', '#09f6ff')
>>> nodes_data = dflx_graph.nxgraph().nodes.data()
>>> wind_node = [n[1] for n in nodes_data if
...     n[0].subtag == "wind" and
...     n[0].region == "DE01"
... ][0]
>>> wind_node["label"]
'source_volatile_wind_DE01'
>>> wind_node['bg_color']
'#6a6a72'
>>> wind_node["type"]
'Source'
>>> my_colors = {"Source": {"bg": "#996967", "fg": "#000000"}}
>>> dflx_graph.color_nodes_by_type(my_colors)
>>> nodes_data = dflx_graph.nxgraph().nodes.data()
>>> wind_node = [n[1] for n in nodes_data if
...     n[0].subtag == "wind" and
...     n[0].region == "DE01"
... ][0]
>>> wind_node['bg_color']
'#996967'
```

deflex.DeflexGraph.write

`DeflexGraph.`**`write`**(*filename*, ***kwargs*)

Write the graph into a .graphml file.

Parameters **`filename`** (*`str`*) – Path of the output file e.g. /my/special/path/mygraph.graphml

Other Parameters **weight_exponent** (*int*) – Shift the decimal point: $weight = weight \cdot 10^{weight\_exponent}$

Examples

```
>>> import os
>>> from deflex import fetch_test_files
>>> from deflex import restore_results
>>> from deflex import DeflexGraph
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> my_results = restore_results(fn)
>>> dflx_graph = DeflexGraph(my_results)
>>> fn_out = fn.replace(".dflx", "_graph.graphml")
>>> dflx_graph.write(fn_out, weight_exponent=-3)
>>> os.stat(fn_out).st_size > 0
True
>>> os.remove(fn_out)
```

deflex.DeflexGraph.color_edges_by_weight

`DeflexGraph.`**`color_edges_by_weight`**(*cmap='cool'*, *max_weight=None*)

Color all edges by their weight using a matplotlib color map (cmap). If no maximum weight is give the highest weight is used.

Parameters

- **`cmap`** (*`str`*) – Name of the matplotlib color map.
- **`max_weight`** (*`numeric`*) – The maximum for the normalisation of the weights. All number above the max_weight will get the color of the maximum. If no value is given the maximum weight of all edges will used.

Examples

```
>>> from deflex import fetch_test_files
>>> from deflex import restore_results
>>> from deflex import DeflexGraph
>>> from matplotlib.cm import get_cmap
>>>
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> my_results = restore_results(fn)
>>> dflx_graph = DeflexGraph(my_results)
>>> dflx_graph.color_edges_by_weight(cmap="rainbow", max_weight=80)
>>> edges = dflx_graph.edges
>>> bus = [edg for edg in edges if "wind" in edg.nodes[0].label][0]
>>> getattr(bus, "color")
'#ff0000'
>>> w = bus.weight
>>> int(w)
336973
>>> rgb2hex(get_cmap("rainbow")(w))
'#ff0000'
```

deflex.DeflexGraph.color_nodes_by_type

`DeflexGraph.`**`color_nodes_by_type`**(*colors*, *use_name=True*)

Color all nodes in a specific color according to their class.

Use the *group_nodes_by_type()* method to get all existing types. Now a color can be assigned to every type using a color dictionary. If no color is defined for an existing class the default color is used. By default the name of each class is used.

Parameters **colors** (*dict*) – The dictionary needs to have the class (name of class) as keys and a color dictionary as value. The color dictionary has two keys, "fg" for the foreground color (font color) and "bg" for the background color (fill color) and the color as value. The color has to be in the hexadecimal style.

use_name [bool] Use the name of the class instead of the class as key. If the class is used, the classes also have to be used for the colors as key.

Examples

```
>>> from deflex import fetch_test_files
>>> from deflex import restore_results
>>> from deflex import DeflexGraph
>>> from oemof import solph
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> my_results = restore_results(fn)
>>> dflx_graph = DeflexGraph(my_results)
>>> my_colors = {
...     "Bus": {"bg": "#00ff11", "fg": "#000000"},
...     "GenericStorage": {"bg": "#efb507", "fg": "#000000"},
...     "Transformer": {"bg": "#94221d", "fg": "#000000"},
...     "Source": {"bg": "#996967", "fg": "#000000"},
...     "Sink": {"bg": "#31306e", "fg": "#ffffff"},
... }
>>> dflx_graph.color_nodes_by_type(my_colors)
>>> bus = [a for a in dflx_graph.nodes if isinstance(a, solph.Bus)][0]
>>> getattr(bus, "bgcolor")
'#00ff11'
>>> sorted(set([a.bgcolor for a in dflx_graph.nodes]))
['#00ff11', '#31306e', '#94221d', '#996967', '#efb507']
>>> sorted(set([a.fgcolor for a in dflx_graph.nodes]))
['#000000', '#ffffff']
>>> my_colors = {
...     solph.Bus: {"bg": "#00ff11", "fg": "#000000"},
...     "GenericStorage": {"bg": "#efb507", "fg": "#000000"},
...     "Transformer": {"bg": "#94221d", "fg": "#000000"},
...     solph.Source: {"bg": "#996967", "fg": "#000000"},
...     solph.Sink: {"bg": "#31306e", "fg": "#ffffff"},
... }
>>> dflx_graph.color_nodes_by_type(my_colors, use_name=False)
>>> bus = [a for a in dflx_graph.nodes if isinstance(a, solph.Bus)][0]
>>> getattr(bus, "bgcolor")
'#00ff11'
>>> sorted(set([a.bgcolor for a in dflx_graph.nodes]))
['#00ff11', '#31306e', '#6a6a72', '#996967']
>>> sorted(set([a.fgcolor for a in dflx_graph.nodes]))
['#000000', '#ffffff']
```

deflex.DeflexGraph.color_nodes_by_substring

`DeflexGraph.`**`color_nodes_by_substring`**`(`*colors*`)`

Color all nodes in a specific color according to a given substring. A color can be assigned to every substring using a dictionary with the substrings as key an the color dictionary as value. The color dictionary needs to have the keys "fg" for the foreground color (font color) and "bg" for the background color (fill color). The color has to be in the hexadecimal style. Each substring key will be compared with the label of the node as string. If no substring match the default node color is used. If more than one substring is within a label the last match will overwrite the previous matches.

Parameters **`colors`** (*`dict`*) – The dictionary needs to have the substring as keys and a color dictionary as value. The color dictionary has two keys, "fg" for the foreground color (font color) and "bg" for the background color (fill color) and the color as value. The color has to be in the hexadecimal style.

Examples

```
>>> from deflex import fetch_test_files
>>> from deflex import restore_results
>>> from deflex import DeflexGraph
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> my_results = restore_results(fn)
>>> dflx_graph = DeflexGraph(my_results)
>>> my_colors = {
...     "H2": {"bg": "#00ff11", "fg": "#000000"},
...     "electricity": {"bg": "#efb507", "fg": "#000000"},
...     "bioenergy": {"bg": "#063313", "fg": "#ffffff"},
... }
>>> dflx_graph.color_nodes_by_substring(my_colors)
>>> mynode = [n for n in dflx_graph.nodes if "bioenergy" in n.label][0]
>>> getattr(mynode, "bgcolor")
'#063313'
>>> sorted(set([n.bgcolor for n in dflx_graph.nodes]))
['#00ff11', '#063313', '#6a6a72', '#efb507']
>>> sorted(set([n.fgcolor for n in dflx_graph.nodes]))
['#000000', '#ffffff']
```

deflex.DeflexGraph.group_nodes_by_type

`DeflexGraph.`**`group_nodes_by_type`**`(`*use_name=False*`)`

Group all nodes by types returning a dictionary with the types or the name of the types as keys and the list of nodes as value.

The keys of the returning dictionary are the classes (or name of the classes) the values are lists with nodes of the corresponding class.

Parameters **`use_name`** (*`bool`*) – Use the name of the class instead of the class as key.

Returns **All nodes sorted by their type**

Return type dict

Examples

```
>>> from deflex import fetch_test_files
>>> from deflex import restore_results
>>> from deflex import DeflexGraph
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> my_results = restore_results(fn)
>>> dflx_graph = DeflexGraph(my_results)
>>> sorted(dflx_graph.group_nodes_by_type(use_name=True))
['Bus', 'GenericStorage', 'Sink', 'Source', 'Transformer']
>>> list(dflx_graph.group_nodes_by_type(use_name=False))[0].__mro__[-2]
<class 'oemof.network.network.Node'>
>>> sources = dflx_graph.group_nodes_by_type(use_name=True)["Source"]
>>> sorted(sources)[-1].label
Label(cat='source', tag='volatile', subtag='wind', region='DE03')
```

4.3.3 Analyse cycles

deflex.Cycles(results[, storages, lines, digits])	Detect all simple cycles in the directed graph.
deflex.Cycles.cycles	Get all cycles of the model.
deflex.Cycles.used_cycles	Get all cycles from a list of cycles that are used.
deflex.Cycles.suspicious_cycles	Get all cycles from a list of cycles that are suspicious.
deflex.Cycles.get_suspicious_time_steps()	Detect the time steps of a cycle in which all flows are non-zero.
deflex.Cycles.print()	Print an overview of the cycles.
deflex.Cycles.details()	Print out a more detailed overview over the existing cycles.

deflex.Cycles

class deflex.**Cycles**(*results*, *storages=True*, *lines=True*, *digits=10*)

Detect all simple cycles in the directed graph.

Furthermore, get the flows of each cycle as pandas.DataFrame. For a large number of cycles getting the values may take a while so check the *simple_cycles* attribute first and consider setting *storages* and *lines* to *False*.

Cycles are a list of nodes with a flow between one node and the following node in the list and a flow from the last node of the lsit to the first node. Therefore, the number of nodes equals the number of flows.

Parameters

- **results** (*dict*) – A valid deflex results dictionary.
- **storages** (*bool*) – Storages are always cycles and you may want to exclude them from the results setting *storages=False*. Nevertheless, sometimes storages are charged and discharged in one time step, which indicates a modelling problem. To detect such behaviour *storages* should be *True*. (default: True)
- **lines** (*bool*) – Transmission lines will create multiple cycles especially in models with a high number of regions and line. Setting *lines* to *False* will exclude cycles that are caused by lines. Cycles with e.g. an electrolyses in one region and a H2 power plant in another will cause a hydrogen-electricity cycle. In this cycle is a transmission line include but this cycle will not(!) be excluded if *lines=False*. (default: True)
- **digits** (*int*) – To detect used or critical cycles the flows are rounded to avoid a detection

for very small flow values. Use *digits* to define the number of digits to be rounded. A high number will make the detection very sensitive. (default: 10)

name
Name of the cycle object.

Type str

simple_cycles
A list of all cycles. Each cycle is a list of nodes.

Type list of lists

Examples

```
>>> from deflex import restore_results, fetch_test_files
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> c = Cycles(restore_results(fn), storages=True, lines=True)
>>> len(list(c.simple_cycles))
9
>>> c = Cycles(restore_results(fn), storages=False, lines=True)
>>> len(list(c.simple_cycles))
7
>>> c = Cycles(restore_results(fn), storages=False, lines=False)
>>> len(list(c.simple_cycles))
2
```

__init__(*results*, *storages=True*, *lines=True*, *digits=10*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(results[, storages, lines, digits])	Initialize self.
`details`()	Print out a more detailed overview over the existing cycles.
`get_suspicious_time_steps`()	Detect the time steps of a cycle in which all flows are non-zero.
`print`()	Print an overview of the cycles.

Attributes

`cycles`	Get all cycles of the model.
`suspicious_cycles`	Get all cycles from a list of cycles that are suspicious.
`used_cycles`	Get all cycles from a list of cycles that are used.

deflex.Cycles.cycles

Cycles.**cycles**
Get all cycles of the model.

Cycles are a list of nodes with a flow between one node and the following node in the list and a flow from the last node of the lsit to the first node. Therefore, the number of nodes equals the number of flows.

Returns

Return type list of pandas.DataFrame

Examples

```
>>> from deflex import restore_results, fetch_test_files, Cycles
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> cy = Cycles(restore_results(fn), storages=True, lines=True)
>>> len(cy.cycles)
9
>>> len(cy.used_cycles)
2
>>> type(cy.used_cycles[0])
<class 'pandas.core.frame.DataFrame'>
```

deflex.Cycles.used_cycles

`Cycles.`**`used_cycles`**

Get all cycles from a list of cycles that are used.

Cycles are not in use if one flow of the cycle is zero for all time steps.

Returns

Return type list of pandas.DataFrame

Examples

```
>>> from deflex import restore_results, fetch_test_files, Cycles
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> cy = Cycles(restore_results(fn), storages=True, lines=True)
>>> len(cy.cycles)
9
>>> len(cy.used_cycles)
2
>>> type(cy.used_cycles[0])
<class 'pandas.core.frame.DataFrame'>
```

deflex.Cycles.suspicious_cycles

`Cycles.`**`suspicious_cycles`**

Get all cycles from a list of cycles that are suspicious.

Suspicious cycles are cycles that have a non-zero value in all flows within one time step.

One can detect all cycles and drop the unsuspicious cycles to get only the suspicious ones. A suspicious cycle indicates a problem in the model design, so one should have a closer look at all these cycles. A typical example for such cycles are storages that a charged and discharged in one time step. In some rare cases suspicious cycles are fine.

Examples

```
>>> from deflex import restore_results, fetch_test_files, Cycles
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> cy = Cycles(restore_results(fn), storages=True, lines=True)
>>> len(list(cy.simple_cycles))
9
>>> len(cy.suspicious_cycles)
0
```

deflex.Cycles.get_suspicious_time_steps

`Cycles.get_suspicious_time_steps()`

Detect the time steps of a cycle in which all flows are non-zero.

Returns **One table for each cycle with all suspicious rows**

Return type list

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_suspicious_modelling.dflx")
>>> my_results = dflx.restore_results(fn)
>>> c = Cycles(my_results)
>>> len(list(c.simple_cycles))
7
>>> len(c.used_cycles)
1
>>> len(c.suspicious_cycles)
1
>>> c.get_suspicious_time_steps()[0].iloc[5]
0_from_storage_electricity_battery_DE01    317596.81
1_from_electricity_all_all_DE01            289581.37
Name: 2022-01-01 05:00:00, dtype: float64
```

deflex.Cycles.print

`Cycles.print()`

Print an overview of the cycles.

Examples

```
>>> from deflex import restore_results, fetch_test_files, Cycles
>>> fn = fetch_test_files("de03_fictive.dflx")
>>> cy = Cycles(restore_results(fn), storages=True, lines=True)
>>> cy.print()
*** Cycle object of scenario: de03_fictive_test ***
<BLANKLINE>
Number of cycles: 9
Number of used cycles: 2
Number of critical cycles: 0
<BLANKLINE>
```

deflex.Cycles.details

`Cycles.details()`

Print out a more detailed overview over the existing cycles.

4.3.4 Basic results processing

`deflex.get_all_results`(results)	Get all results from a computed deflex scenario.
`deflex.nodes2table`(results[, no_sums])	Get a table with all nodes as a MultiIndex with the sum of their in an out flows.
`deflex.solver_results2series`(results)	Get the meta results from the solver.
`deflex.fetch_dual_results`(results[, bus, ...])	Collect all the results of the dual variables.
`deflex.meta_results2series`(results)	Get meta results as a pandas.Series
`deflex.get_time_index`(results)	Get the time index of the model.
`deflex.calculate_key_values`(results[, ...])	Get time series of typical key values.

deflex.get_all_results

`deflex.get_all_results(`*results*`)`

Get all results from a computed deflex scenario.

The results will be returned as a dictionary of pandas.DataFrame that can be stored in the xlsx or csv format using *dict2file*. This function can be used to transfer the results to another programming language or an external tool.

Parameters **results** (*dict*) – A valid deflex results dictionary.

Returns

Return type dict

Examples

```
>>> import os
>>> import shutil
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> all_results = get_all_results(my_results)
>>> sorted(list(all_results.keys()))[:4]
['commodity', 'components', 'electricity', 'heat_decentralised']
>>> sorted(list(all_results.keys()))[-5:]
['heat_decentralised', 'heat_district', 'meta', 'mobility', 'solver']
>>> fn_out = fn.replace(".dflx", "_all_results.csv")
>>> dflx.dict2file(all_results, fn_out, "csv", drop_empty_columns=True)
>>> my_bool = []
>>> for key in all_results.keys():
...     fn_test = os.path.join(fn_out, key + ".csv")
...     my_bool.append(os.path.isfile(fn_test))
>>> my_bool
[True, True, True, True, True, True, True, True]
>>> shutil.rmtree(fn_out)
```

deflex.nodes2table

deflex.**nodes2table**(*results*, *no_sums=False*)

Get a table with all nodes as a MultiIndex with the sum of their in an out flows.

The index contains the following levels: class, category, tag, subtag, region

The sums can be found in the columns "in" and "out".

Parameters

- **results** (*dict*) – Deflex results dictionary.
- **no_sums** (*bool*) – Set to False to get an empty DataFrame with no sums (default: True)

Returns **Table with all nodes and sums**

Return type pandas.DataFrame

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> all_nodes = nodes2table(my_results)
>>> len(all_nodes)
226
>>> all_nodes.to_csv("your/path/file.csv")  # doctest: +SKIP
```

deflex.solver_results2series

deflex.**solver_results2series**(*results*)

Get the meta results from the solver.

The keys in the first index level are:

- Problem
- Solution
- Solver
- Solver Black box
- Solver Branch and bound

Parameters **results** (*dict*) – A valid deflex results dictionary.

Returns

Return type pandas.Series

Example

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_heat.dflx")
>>> my_results = dflx.restore_results(fn)
>>> slvr = solver_results2series(my_results)
```

(continues on next page)

(continued from previous page)

```
>>> list(slvr.index.get_level_values(0).unique())[:4]
['Problem', 'Solution', 'Solver', 'Solver Black box']
>>> round(slvr["Solver", "Time"],5)
0.07627
>>> int(slvr["Solution", "Objective"])
7516285616
```

deflex.fetch_dual_results

`deflex.`**`fetch_dual_results`**`(`*results*, *bus=None*, *exclude_commodities=True*`)`

Collect all the results of the dual variables.

A bus can be passed to get only the dual variables of this specific bus, otherwise the results of the dual variables of all buses are collected. The variables of the commodity buses can be excluded using the *exclude_commodities* parameter.

Parameters

- **`results`** (*`dict`*) – A valid deflex results dictionary.
- **`bus`** (*`oemof.network.Bus`*) – An existing Bus of the deflex model results.
- **`exclude_commodities`** (*`bool`*) – Exclude the results of the commodity buses.

Returns

Return type pandas.Series

deflex.meta_results2series

`deflex.`**`meta_results2series`**`(`*results*`)`

Get meta results as a pandas.Series

deflex.get_time_index

`deflex.`**`get_time_index`**`(`*results*`)`

Get the time index of the model.

deflex.calculate_key_values

`deflex.`**`calculate_key_values`**`(`*results*, *ignore_chp=True*`)`

Get time series of typical key values.

- marginal costs
- highest emission
- lowest emission
- marginal costs power plant
- emission of marginal costs power plant

Parameters

- **`results`** (*`dict`*) – Deflex results dictionary.

- **ignore_chp** (*bool*) – Set False to include the chp-plants (default: True).

Returns

Return type pandas.DataFrame

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> df = calculate_key_values(my_results, ignore_chp=False)
>>> list(df.columns)[:3]
['marginal costs', 'highest emission', 'lowest emission']
>>> row = df.iloc[24]
>>> row.pop("marginal costs power plant").label
Label(cat='chp plant', tag='bioenergy', subtag='bioenergy', region='DE01')
>>> row
marginal costs                                47.573824
highest emission                                   1.01
lowest emission                                     0.0
emission of marginal cost power plant          0.016992
Name: 2022-01-02 00:00:00, dtype: object
>>> min_mc = df["marginal costs"].min()
>>> max_mc = df["marginal costs"].max()
>>> print("{0} - {1}".format(round(min_mc, 2), round(max_mc, 2)))
47.57 - 65.35
>>> df = calculate_key_values(my_results, ignore_chp=True)
>>> row = df.iloc[45]
>>> str(row.pop("marginal costs power plant").label)
'power-plant_natural-gas_06_natural-gas_DE01'
>>> row
marginal costs                                46.230384
highest emission                               1.299035
lowest emission                                     0.0
emission of marginal cost power plant          0.335559
Name: 2022-01-02 21:00:00, dtype: object
>>> min_mc = df["marginal costs"].min()
>>> max_mc = df["marginal costs"].max()
>>> print("{0} - {1}".format(round(min_mc, 2), round(max_mc, 2)))
29.97 - 47.58
```

4.3.5 Advanced results processing

deflex.group_buses(buses, fields)	Group buses by parts of the label.
deflex.fetch_converter_parameters(results, …)	Fetch relevant parameters of every Transformer of the energy system.
deflex.fetch_attributes_of_commodity_sources(results)	Get the attributes of the commodity sources.
deflex.get_combined_bus_balance(results[, …])	Combine different buses of the same type.
deflex.get_converter_balance(results[, cat, …])	Get the balance around the converters of the system.

deflex.group_buses

deflex.**group_buses**(*buses*, *fields*)

Group buses by parts of the label.

Parameters

- **buses** (*list*) – Buses to group.
- **fields** (*list*) – Fields of the label to group the buses. Valid labels are *cat*, *tag*, *subtag*, *region*.

Returns Grouped buses

Return type dict

Examples

```
>>> import deflex as dflx
>>> from oemof.network.network import Bus
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> mybuses = set([r[0] for r in my_results["main"].keys()
...     if isinstance(r[0], Bus)])
>>> sorted(dflx.group_buses(mybuses, ["cat", "tag", "subtag"]).keys())[:2]
[('commodity', 'all', 'H2'), ('commodity', 'all', 'bioenergy')]
>>> sorted(dflx.group_buses(mybuses, ["cat"]).keys())[:4]
[('commodity',), ('electricity',), ('heat',), ('mobility',)]
>>> c_buses = dflx.group_buses(mybuses, ["cat"])[('commodity',)]
>>> sorted(c_buses)[0].label
Label(cat='commodity', tag='all', subtag='H2', region='DE')
>>> len(c_buses)
10
>>> for bu in sorted(c_buses)[:3]:
...     print(repr(bu.label))
Label(cat='commodity', tag='all', subtag='H2', region='DE')
Label(cat='commodity', tag='all', subtag='bioenergy', region='DE01')
Label(cat='commodity', tag='all', subtag='bioenergy', region='DE02')
```

deflex.fetch_converter_parameters

deflex.**fetch_converter_parameters**(*results*, *transformer*)

Fetch relevant parameters of every Transformer of the energy system.

Returns

Return type pandas.DataFrame

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> power_plants = [
...     bk[1] for bk in my_results["main"].keys()
...     if bk[1] is not None
```

(continues on next page)

(continued from previous page)

```
...     and bk[1].label.subtag == "natural gas"
...     and isinstance(bk[1], solph.Transformer)
... ]
>>> table = fetch_converter_parameters(my_results, power_plants)
>>> power_plant = table.iloc[5].dropna()
>>> power_plant.name = power_plant.pop("label_str")
>>> power_plant
allocation method                    electricity
category                             power plant
efficiency, electricity                    0.311
emission, fuel                             0.201
fuel                             natural gas, DE
specific_costs_electricity             89.041801
specific_emission_electricity           0.646302
variable costs, fuel                      27.692
Name: power-plant_natural-gas_031_natural-gas_DE01, dtype: object
>>> power_plant = table.iloc[0].dropna()
>>> power_plant.name = power_plant.pop("label_str")
>>> power_plant
allocation method                        finnish
category                               chp plant
efficiency, electricity                     0.25
efficiency, heat                            0.41
emission, fuel                             0.201
fuel                             natural gas, DE
specific_costs_electricity                 57.96
specific_costs_heat                         32.2
specific_emission_electricity           0.420698
specific_emission_heat                  0.233721
variable costs, fuel                      27.692
Name: chp-plant_natural-gas_natural-gas_DE01, dtype: object
```

deflex.fetch_attributes_of_commodity_sources

deflex.**fetch_attributes_of_commodity_sources**(*results*)

Get the attributes of the commodity sources.

Transformers like power plants are connected to commodity buses. This function can be used to get specific emission or the variable costs of the connected commodity source. Use the *to_node* column to find the data row of the commodity Bus of the Transformer.

Parameters **results** (*dict*) – Deflex results dictionary.

Returns **The attributes of all commodities**

Return type pandas.DataFrame

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> cdf = dflx.fetch_attributes_of_commodity_sources(my_results)
>>> hard_coal = cdf.loc["hard coal", "DE"]
>>> hard_coal.pop("from_node").label
```

(continues on next page)

(continued from previous page)

```
Label(cat='source', tag='commodity', subtag='hard coal', region='DE')
>>> hard_coal.pop("to_node").label
Label(cat='commodity', tag='all', subtag='hard coal', region='DE')
>>> hard_coal
emission                    0.337
nominal_value                 NaN
summed_max                    NaN
max                           1.0
min                           0.0
negative_gradient_costs       0.0
positive_gradient_costs       0.0
variable_costs             19.944
Name: (hard coal, DE), dtype: object
>>> flow_to_power_plant = [
...     bk for bk in my_results["main"].keys()
...     if bk[1] is not None
...     and bk[1].label.cat == "power plant"
...     and bk[1].label.subtag == "natural gas"
... ][0]
>>> float(cdf.loc[cdf.to_node == flow_to_power_plant[0]].emission)
0.201
```

deflex.get_combined_bus_balance

`deflex.`**`get_combined_bus_balance`**(*results*, *cat=None*, *tag=None*, *subtag=None*, *region=None*)

Combine different buses of the same type.

The combined buses can be restricted by the label fields (cat, tag, subtag, region). Only buses with the same label fields will be combined.

Parameters

- **results** (*dict*) – Deflex results dictionary.
- **cat** (*str*) – Category of the buses.
- **tag** (*str*) – Tag of the buses.
- **subtag** (*str*) – Subtag of the buses.
- **region** (*str*) – Region of the buses

Returns

Return type pandas.DataFrame

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> get_combined_bus_balance(my_results, cat="electricity")["out"].columns
MultiIndex([('decentralised heat',    'heat pump',   'heat pump', 'DE02'),
            ('electricity demand',  'electricity',         'all', 'DE01'),
            ('electricity demand',  'electricity',         'all', 'DE02'),
            (            'excess',  'electricity',         'all', 'DE01'),
            (            'excess',  'electricity',         'all', 'DE02'),
```

(continues on next page)

(continued from previous page)

```
            (            'excess',  'electricity',         'all', 'DE03'),
            (    'fuel converter',  'electricity', 'electricity', 'DE01'),
            (    'fuel converter',  'electricity', 'electricity', 'DE02'),
            (              'line',  'electricity',        'DE01', 'DE02'),
            (              'line',  'electricity',        'DE01', 'DE03'),
            (              'line',  'electricity',        'DE02', 'DE01'),
            (              'line',  'electricity',        'DE02', 'DE03'),
            (              'line',  'electricity',        'DE03', 'DE01'),
            (              'line',  'electricity',        'DE03', 'DE02'),
            (   'other converter', 'Electrolysis', 'electricity',   'DE'),
            (           'storage',  'electricity',     'battery', 'DE01'),
            (           'storage',  'electricity',        'phes', 'DE01')],
           )
>>> get_combined_bus_balance(
...     my_results, cat="electricity", region="DE03")["out"].columns
MultiIndex([('excess', 'electricity',  'all', 'DE03'),
            (  'line', 'electricity', 'DE03', 'DE01'),
            (  'line', 'electricity', 'DE03', 'DE02')],
           )
```

deflex.get_converter_balance

deflex.**get_converter_balance**(*results*, *cat=None*, *tag=None*, *subtag=None*, *region=None*)

Get the balance around the converters of the system.

The converters can be restricted by the label fields (cat, tag, subtag, region). Only converters with the same label fields will be shown.

Parameters

- **results** (*dict*) – Deflex results dictionary.
- **cat** (*str*) – Category of the buses.
- **tag** (*str*) – Tag of the buses.
- **subtag** (*str*) – Subtag of the buses.
- **region** (*str*) – Region of the buses

Returns

Return type pandas.DataFrame

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de03_fictive.dflx")
>>> my_results = dflx.restore_results(fn)
>>> hc49 = get_converter_balance(
...     my_results, cat="power plant", tag="hard coal_049").sum()
>>> round(float((hc49["out"] / hc49["in"])), 2)
0.49
```

4.3.6 Tools for Electricity models

deflex.merit_order_from_scenario(scenario[, ...])	Create a merit order from a deflex scenario.
deflex.merit_order_from_results(result)	Create a merit order from deflex results.

deflex.merit_order_from_scenario

deflex.**merit_order_from_scenario**(*scenario*, *with_downtime=True*, *with_co2_price=True*)

Create a merit order from a deflex scenario.

Parameters

- **scenario** (*deflex.Scenario*) – Path of the directory where the csv files of the scenario are located.
- **with_downtime** (*bool*) – Use down time factor to reduce the installed capacity.
- **with_co2_price** (*bool*) – Consider the CO2 price to calculate the merit order.

Returns

Return type pandas.DataFrame

Examples

```
>>> import os
>>> import deflex as dflx
>>> my_path = dflx.fetch_test_files("de02_no-heat_csv")
>>> my_sc = dflx.DeflexScenario()
>>> mo1 = dflx.merit_order_from_scenario(my_sc.read_csv(my_path))
>>> round(mo1.capacity_cum.iloc[-1], 4)
86.7028
>>> round(mo1.capacity.sum(), 1)
86702.8
>>> round(mo1.loc[("DE01", "natural gas - 0.55"), "costs_total"], 2)
49.37
>>> mo2 = merit_order_from_scenario(my_sc.read_csv(my_path),
...                                 with_downtime=False)
>>> int(round(mo2.capacity.sum(), 0))
101405
>>> mo3 = merit_order_from_scenario(my_sc.read_csv(my_path),
...                                 with_co2_price=False)
>>> round(mo3.loc[("DE01", "natural gas - 0.55"), "costs_total"], 2)
43.58
```

deflex.merit_order_from_results

deflex.**merit_order_from_results**(*result*)

Create a merit order from deflex results.

Parameters **result** (*dict*) – A deflex results dictionary.

Returns

Return type pandas.DataFrame

Examples

```
>>> import deflex as dflx
>>> fn = dflx.fetch_test_files("de02_no-heat.dflx")
>>> my_results = dflx.restore_results(fn)
>>> a = merit_order_from_results(my_results)
```

4.3.7 Geometry examples for plotting

deflex.deflex_geo(rmap)	Fetch default deflex geometries as a named tuple with the following fields:
deflex.divide_off_and_onshore(regions)	Sort regions into onshore and offshore regions (Germany).

deflex.deflex_geo

deflex.**deflex_geo**(*rmap*)

Fetch default deflex geometries as a named tuple with the following fields:

- polygons
- lines
- labels
- line_labels

Note that some fields might be None for some region sets.

Parameters **rmap** (*str*) – Name of the deflex map. Possible values are: de01, de02, de17, de21, de22

Returns

Return type namedtuple

Examples

```
>>> de02 = deflex_geo("de02")
>>> list(de02.polygons.index)
['DE01', 'DE02']
>>> p = de02.labels.loc["DE01"].geometry
>>> p.x, p.y
(10.0, 51.6)
>>> de02.lines.index
Index(['DE01-DE02'], dtype='object', name='name')
>>> de02.line_labels.iloc[0]
gid                           246
rotation                      -42
geometry    POINT (7.61 53.78)
Name: DE01-DE02, dtype: object
>>> de01 = deflex_geo("de01")
>>> print(de01.lines)
None
```

deflex.divide_off_and_onshore

deflex.**divide_off_and_onshore**(*regions*)

Sort regions into onshore and offshore regions (Germany).

A namedtuple with two list of regions ids will be returned. Fetch the *onshore* and *offshore* attribute of the named tuple to get the list.

Parameters **regions** (*GeoDataFrame*) – A region set with the region id in the index.

Returns

Return type namedtuple

Examples

```
>>> reg=deflex_regions('de02')
>>> divide_off_and_onshore(reg).onshore
['DE01']
>>> reg=deflex_regions('de21')
>>> divide_off_and_onshore(reg).offshore
['DE19', 'DE20', 'DE21']
```

4.3.8 General tools

deflex.dict2file(tables, path[, filetype, …])	**param tables**
deflex.use_logging(**kwargs)	

deflex.dict2file

deflex.**dict2file**(*tables*, *path*, *filetype='xlsx'*, *drop_empty_columns=False*)

Parameters

- **tables** –
- **path** –
- **filetype** –
- **drop_empty_columns** –

deflex.use_logging

deflex.**use_logging**(***kwargs*)

4.3.9 CHP allocation tools

deflex.allocate_fuel_deflex(method, eta_e, …)	Allocate the fuel input of chp plants to the two output flows.

Continued on next page

Table 20 – continued from previous page

`deflex.allocate_fuel`(method, eta_e, eta_th, . . .)	Allocate the fuel input of chp plants to the two output flows: heat and electricity.
`deflex.efficiency_method`(eta_e, eta_th)	Efficiency Method - a method to allocate the fuel input of chp plants to the two output flows: heat and electricity
`deflex.exergy_method`(eta_e, eta_th, eta_c)	Exergy Method or Carnot Method- a method to allocate the fuel input of chp plants to the two output flows: heat and electricity
`deflex.finnish_method`(eta_e, eta_th, . . .)	Alternative Generation or Finnish Method - a method to allocate the fuel input of chp plants to the two output flows: heat and electricity
`deflex.iea_method`(eta_e, eta_th)	IEA Method (International Energy Association - a method to allocate the fuel input of chp plants to the two output flows: heat and electricity

deflex.allocate_fuel_deflex

`deflex.`**`allocate_fuel_deflex`**(*method*, *eta_e*, *eta_th*)

Allocate the fuel input of chp plants to the two output flows.

In contrast to `allocate_fuel()` default parameter from the config file (deflex.ini) are used.

To change the default parameters create a deflex.ini file in $HOME/.deflex and add the following section:

[chp] eta_c = 0.555 eta_e_ref = 0.5 eta_th_ref = 0.9

This will overwrite the default values from deflex and use them as user default values. Lines with values that are not needed in the chosen method can be removed.

The following methods are available:

- Alternative Generation or Finnish method -> `finnish_method()`
- Exergy method or Carnot method -> `exergy_method()`
- IEA method -> `iea_method()`
- Efficiency method -> `efficiency_method()`

Parameters

- **`method`** (*str*) – The method to allocate the output flows of chp plants: alternative_generation, carnot, efficiency, electricity, exergy, finnish, heat, iea
- **`eta_e`** (*numeric*) – The efficiency of the electricity production in the chp plant.
- **`eta_th`** (*numeric*) – The efficiency of the heat output in the chp plant.

Returns **The fuel factors of the output flows (heat/electricity)**

Return type namedtuple

Examples

```
>>> a = allocate_fuel_deflex("efficiency", eta_e=0.3, eta_th=0.5)
>>> round(a.electricity, 2)
2.08
>>> round(a.heat, 2)
```

(continues on next page)

(continued from previous page)

```
0.75
>>> a = allocate_fuel_deflex("electricity", eta_e=0.3, eta_th=0.5)
>>> round(a.electricity, 2)
3.33
>>> a.heat
0.0
>>> a = allocate_fuel_deflex("exergy", eta_e=0.3, eta_th=0.5)
>>> round(a.electricity, 2)
1.73
>>> round(a.heat, 2)
0.96
>>> a = allocate_fuel_deflex("finnish", eta_e=0.3, eta_th=0.5)
>>> round(a.electricity, 2)
1.73
>>> round(a.heat, 2)
0.96
>>> a = allocate_fuel_deflex("heat", eta_e=0.3, eta_th=0.5)
>>> a.electricity
0.0
>>> a.heat
2.0
>>> a = allocate_fuel_deflex("iea", eta_e=0.3, eta_th=0.5)
>>> round(a.electricity, 2)
1.25
>>> round(a.heat, 2)
1.25
```

deflex.allocate_fuel

`deflex.`**`allocate_fuel`**(*method*, *eta_e*, *eta_th*, ***kwargs*)

Allocate the fuel input of chp plants to the two output flows: heat and electricity.

Use *`allocate_fuel_deflex()`* if you want to use the default values of the config file or if you want to define your own default values.

The following methods are available:

- Alternative Generation or Finnish method -> *`finnish_method()`*
- Exergy method or Carnot method -> *`exergy_method()`*
- IEA method -> *`iea_method()`*
- Efficiency method -> *`efficiency_method()`*

The sum of the allocation factors of both flows is always one: $\alpha_{th} + \alpha_{el} = 1$

The fuel factor is the allocation factor devided by the efficiency:

$$f_{fuel,el} = \frac{\alpha_{el}}{\eta_{el}} \qquad f_{fuel,th} = \frac{\alpha_{th}}{\eta_{th}}$$

$f_{fuel,el/th}$:Fuel factor of the electricity/heat flow

$\alpha_{el/th}$: Allocation factor of the electricity/heat flow

$\eta_{el/th}$: Efficiency of the electricity/heat output in the chp plant

Parameters

- **method** (*str*) – The method to allocate the output flows of chp plants: alternative_generation, carnot, efficiency, electricity, exergy, finnish, heat, iea
- **eta_e** (*numeric*) – The efficiency of the electricity production in the chp plant. Mandatory for all functions.
- **eta_th** (*numeric*) – The efficiency of the heat output in the chp plant.Mandatory for all functions.

Other Parameters

- **eta_c** (*numeric*) – The Carnot factor of the heating system. Mandatory in the following functions: exergy
- **eta_e_ref** (*numeric*) – The efficiency of the best power plant available on the market and economically viable in the year of construction of the CHP plant. Mandatory in the following functions: alternative_generation
- **eta_th_ref** (*numeric*) – The efficiency of the best heat plant available on the market and economically viable in the year of construction of the CHP plant. Mandatory in the following functions: alternative_generation

Returns **The fuel factors of the output flows (heat/electricity)**

Return type namedtuple

Examples

```
>>> a = allocate_fuel("efficiency", eta_e=0.3, eta_th=0.5)
>>> round(a.electricity, 2)
2.08
>>> round(a.heat, 2)
0.75
>>> a = allocate_fuel("electricity", eta_e=0.3, eta_th=0.5)
>>> round(a.electricity, 2)
3.33
>>> a.heat
0.0
>>> a = allocate_fuel("exergy", eta_e=0.3, eta_th=0.5, eta_c=0.555)
>>> round(a.electricity, 2)
1.73
>>> round(a.heat, 2)
0.96
>>> a = allocate_fuel("finnish", eta_e=0.3, eta_th=0.5, eta_e_ref=0.5,
...                    eta_th_ref=0.9)
>>> round(a.electricity, 2)
1.73
>>> round(a.heat, 2)
0.96
>>> a = allocate_fuel("heat", eta_e=0.3, eta_th=0.5)
>>> a.electricity
0.0
>>> a.heat
2.0
>>> a = allocate_fuel("iea", eta_e=0.3, eta_th=0.5)
>>> round(a.electricity, 2)
1.25
```

(continues on next page)

(continued from previous page)

```
>>> round(a.heat, 2)
1.25
```

deflex.efficiency_method

`deflex.`**`efficiency_method`**(*eta_e*, *eta_th*)

Efficiency Method - a method to allocate the fuel input of chp plants to the two output flows: heat and electricity

The allocation factor α_{el} of the electricity output is calculated as follows:

$$\alpha_{el} = \frac{\eta_{th}}{\eta_{el} + \eta_{th}}$$

α_{el} : Allocation factor of the electricity flow

η_{el} : Efficiency of the electricity output in the chp plant

η_{th} : Efficiency of the thermal output in the chp plant

Parameters

- **eta_e** (*numeric*) – The efficiency of the electricity production in the chp plant. Mandatory in the following functions: alternative_generation, exergy, iea, efficiency
- **eta_th** (*numeric*) – The efficiency of the heat output in the chp plant. Mandatory in the following functions: alternative_generation, exergy, iea, efficiency

Returns **Allocation factor for the electricity flow**

Return type numeric

Examples

```
>>> round(efficiency_method(0.3, 0.5), 3)
0.625
```

deflex.exergy_method

`deflex.`**`exergy_method`**(*eta_e*, *eta_th*, *eta_c*)

Exergy Method or Carnot Method- a method to allocate the fuel input of chp plants to the two output flows: heat and electricity

The allocation factor α_{el} of the electricity output is calculated as follows:

$$\alpha_{el} = \frac{\eta_{el}}{\eta_{el} + \eta_c \cdot \eta_{th}}$$

α_{el} : Allocation factor of the electricity flow

η_{el} : Efficiency of the electricity output in the chp plant

η_{th} : Efficiency of the thermal output in the chp plant

η_c : Carnot factor of the thermal energy

Parameters

- **eta_e** (*numeric*) – The efficiency of the electricity production in the chp plant. Mandatory in the following functions: alternative_generation, exergy, iea, efficiency

- **eta_th** (*numeric*) – The efficiency of the heat output in the chp plant. Mandatory in the following functions: alternative_generation, exergy, iea, efficiency
- **eta_c** (*numeric*) – The Carnot factor of the heating system. Mandatory in the following functions: exergy

Returns **Allocation factor for the electricity flow**

Return type numeric

Examples

```
>>> round(exergy_method(0.3, 0.5, 0.3), 3)
0.667
```

deflex.finnish_method

deflex.**finnish_method**(*eta_e, eta_th, eta_e_ref, eta_th_ref*)

Alternative Generation or Finnish Method - a method to allocate the fuel input of chp plants to the two output flows: heat and electricity

The allocation factor α_{el} of the electricity output is calculated as follows:

$$\alpha_{el} = \frac{\eta_{el,ref}}{\eta_{el}} \cdot \left(\frac{\eta_{el}}{\eta_{el,ref}} + \frac{\eta_{th}}{\eta_{th,ref}} \right)$$

α_{el} : Allocation factor of the electricity flow

η_{el} : Efficiency of the electricity output in the chp plant

η_{th} : Efficiency of the thermal output in the chp plant

$\eta_{el,ref}$: Efficiency of the reference power plant

$\eta_{th,ref}$: Efficiency of the reference heat plant

Parameters

- **eta_e** (*numeric*) – The efficiency of the electricity production in the chp plant. Mandatory in the following functions: alternative_generation, exergy, iea, efficiency
- **eta_th** (*numeric*) – The efficiency of the heat output in the chp plant. Mandatory in the following functions: alternative_generation, exergy, iea, efficiency
- **eta_e_ref** (*numeric*) – The efficiency of the best power plant available on the market and economically viable in the year of construction of the CHP plant. Mandatory in the following functions: alternative_generation
- **eta_th_ref** (*numeric*) – The efficiency of the best heat plant available on the market and economically viable in the year of construction of the CHP plant. Mandatory in the following functions: alternative_generation

Returns **Allocation factor for the electricity flow**

Return type numeric

Examples

```
>>> round(finnish_method(0.3, 0.5, 0.5, 0.9), 3)
0.519
```

deflex.iea_method

deflex.**iea_method**(*eta_e*, *eta_th*)

IEA Method (International Energy Association - a method to allocate the fuel input of chp plants to the two output flows: heat and electricity

The allocation factor α_{el} of the electricity output is calculated as follows:

$$\alpha_{el} = \frac{\eta_{el}}{\eta_{el} + \eta_{th}}$$

α_{el} : Allocation factor of the electricity flow

η_{el} : Efficiency of the electricity output in the chp plant

η_{th} : Efficiency of the thermal output in the chp plant

Parameters

- **eta_e** (*numeric*) – The efficiency of the electricity production in the chp plant. Mandatory in the following functions: alternative_generation, exergy, iea, efficiency
- **eta_th** (*numeric*) – The efficiency of the heat output in the chp plant. Mandatory in the following functions: alternative_generation, exergy, iea, efficiency

Returns **Allocation factor for the electricity flow**

Return type numeric

Examples

```
>>> round(iea_method(0.3, 0.5), 3)
0.375
```

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

deflex could always use more documentation, whether as part of the official deflex docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/reegis/deflex/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *deflex* for local development:

1. Fork deflex (look for the "Fork" button).
2. Clone your fork locally:

   ```
   git clone git@github.com:YOURGITHUBNAME/deflex.git
   ```

3. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.
4. When you're done making changes run all the checks and docs builder with tox one command:

   ```
   tox
   ```

5. Commit your changes and push your branch to GitHub:

   ```
   git add .
   git commit -m "Your detailed description of your changes."
   git push origin name-of-your-bugfix-or-feature
   ```

6. Submit a pull request through the GitHub website.

5.5 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.6 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.

It will be slower though . . .

5.7 Development

To run all the tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append`
	`tox`

Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER 6

Authors

- Uwe Krien - University of Bremen

CHAPTER 7

Changelog

7.1 0.3.1 (2021-05 ??)

- . . .

7.2 0.3.0 (2021-03-25)

There are many API changes in the v0.3.0 release but it is planned to keep the API more stable from now on.

- Layout of input data changed
- Attributes und function names for adapted.
- Documentation now covers all parts of the package
- Examples and illustrative images were added
- Code were cleaned up
- Tests are more stable, test coverage increased slightly
- Example data (v03) can be downloaded from the deflex OSF page.

Authors

- Uwe Krien
- Pedro Duran

7.3 0.2.0 (2021-01-25)

- Move basic scenario with reegis dependency to new package
- Revise structure

- Add tox tests: pyflake, docs, coverage, tests, link-test, manifest, isort

Authors

- Uwe Krien

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index