

Quickstart

In order to improve the distributability of SecMOD the package code is separated from the handled input data. This allows to use a single installation of SecMOD for several projects or scenarios in multiple working directories.

To get started with SecMOD, the following two steps must be completed:

1. [Installation of SecMOD](#)
2. [Set up a working directory](#)

For the first step follow the instructions for the [installation of SecMOD](#). Afterwards return to [Step 2](#) to set up your first working directory.

Set up a working directory

Create a new directory where you want. Note that this directory will contain all input and results data and can become large in size, depending on the size of your examined model.

1. Open a terminal with an activated Python environment (e.g. Anaconda prompt)!
2. Navigate to your newly created directory in the terminal (e.g. “cd D:/WorkingDirectory”)!
3. Run the command “python -m secmod.setup” in the terminal!

SecMOD will now create the necessary folders for the input data and furthermore copy the sample data provided in the SecMOD repository. Afterwards all necessary data to run SecMOD is available in the working directory. Furthermore, a file called “start.bat” is created at the top level of your working directory. This file allows you to start SecMOD with a simple double click. Depending on

your Python installation, you might need to edit the file (right click > edit) according to the instructions.

If you encounter problems during the automatic setup, please refer to [Troubleshooting](#).

[← Previous](#)

[Next →](#)

© Copyright 2021, Institute of Technical Thermodynamics, RWTH Aachen University

Built with [Sphinx](#) using a [theme](#) provided by [Read the Docs](#).

Installation

The installation of SecMOD is easily done. If you want to edit its source code, you need to follow [Editable Installation](#). If you just want to use the published package, follow [Package Installation](#).

Editable Installation

To edit the source code and therefore contribute to the development of SecMOD, you need to clone the [repository](#) to your local machine **recursively**. Using SSH as authentication to the git server the following command needs to be used:

```
git clone --recurse-submodules -j8 git@git-ce.rwth-aachen.de:l1t/opt/secmod/secmod.git 01-SecMO
```

Note

If you don't know yet how to clone a repository have a look [here](#). Make sure that you have Python already installed on your PC.

After the clone process is completed, open a python terminal e.g. in your IDE (Visual Studio Code, Spyder, ...) or the Anaconda prompt. Afterwards you can install SecMOD in form of your local repository by using the following install command:

```
pip install --user -e PATH
```

Note

The PATH at the end of the command stands for the directory of your local repository. You can either use an absolute path or a path relative to the currently active directory of the terminal.

Your local repository should now be installed, if no errors occurred during the process. You can now proceed to the [quickstart](#) guide to complete your setup.

Package Installation

! Error

SecMOD is not yet packaged and therefore not available through PyPI. Use the [Editable Installation](#) instead.

See [Issue 13](#) for further information about the packaged distribution of SecMOD.

If you don't need to edit the source code of the SecMOD package, but just want to use the modules, you can install the package directly from PyPI using the following command:

```
pip install secmod
```

SecMOD should now be installed, if no errors occurred during the process. You can now proceed to the [quickstart](#) guide to complete your setup.

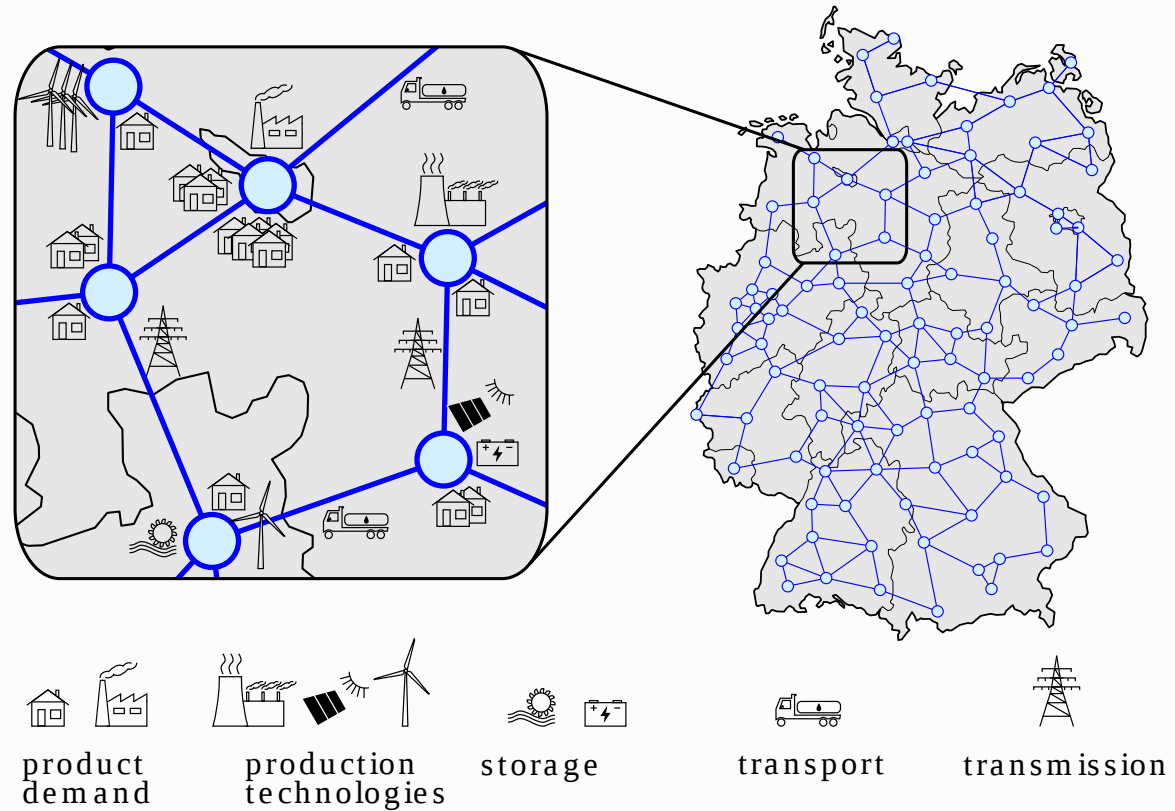
[← Previous](#)[Next →](#)

Data in SecMOD

Please refer to our publication for a detailed description of required data.

Input data

Input data is used to model a given multi-sector system with the desired spatial, technical and temporal resolution. Input data can be added manually in the `sampladata` folder or in some cases be obtained automatically from open webpages, such as the open power system database by the scripts in `sampladata/00-EXTERNAL`.



Units in SecMOD

SecMOD employs unitizing to automatically convert units. All units used in the model must be defined in the sample data file.

Output data and evaluation

The objective of the optimization is to determine an optimal design and operation of the multi-sector system according to the given objective and constraints. The output data of the optimization is saved in the working directory in *01-MODEL-RESULTS/InvestmentModel_year.pickle*.

Usually, the evaluation will start automatically. To start the evaluation without the optimization, type:

```
python -m secmod.evaluation
```

This will open a graphical user interface (GUI). In the user interface, the capacity, product flows, and impacts can be shown for all products in different plot types. Clicking on a capacity will open a second layer to investigate the construction years. Further, all raw results are shown as a table next to the plot. The data can be exported in several formats including tikz, png, xlsx and pdf.

[< Previous](#)[Next >](#)

© Copyright 2021, Institute of Technical Thermodynamics, RWTH Aachen University

Built with [Sphinx](#) using a [theme](#) provided by [Read the Docs](#).

Developer reference

secmod.classes module

```
class secmod.classes.Grid(grid_path: pathlib.Path = None, name: str = None, nodes: pandas.core.frame.DataFrame = None, connections: pandas.core.frame.DataFrame = None) \[source\]
```

Bases: `object`

This class is used to manage grids of nodes and connections.

This class represents a grid, which consists of a name, nodes and connections between these nodes. It can be created from scratch by providing at least a unique name and optionally pandas dataframes for data about the nodes and connections. It can alternatively be created from a path to a directory of an existing grid. The name will then be taken from the name of the grid directory and the node data and connection data will be loaded from .csv-files.

Furthermore the class has a static list of all created instances and a single static variable which represents the selected grid for the optimization.

Parameters

- **grid_path** (*Path*) – A path of the directory of an existing grid definition, which includes a CSV-file with data about nodes and connections in the grid.
- **name** (*str*) – Name for a new grid, if no grid_path is provided. The name has to be unique, since no two grids can have the same name.
- **nodes** (*pd.DataFrame*) – A pandas dataframe which has all the information about the nodes of a new grid, if no grid_path is provided. This dataframe must have the columns ["node","latitude","longitude"].

- **connections** (*pd.DataFrame*) – A pandas dataframe which has all the information about the connections of a new grid, if no `grid_path` is provided. This dataframe must have the columns ["connection","node1","node2"].

grids

Static list of all instanciated grids.

Type

list

_selected

Static reference to a single Grid instance which is selected to be used in the optimization. Can be reached via the `selected()`-method.

Type

[Grid](#)

Raises

- **ValueError** – If no name is provided, if no path is provided as well.
- **FileNotFoundError** – If “nodes.csv” or “connections.csv” does not exist in the provided directory.
- **NotADirectoryError** – If the provided path is no directory.

```
CONNECTION_NODES= {None: ['node1', 'node2']}
```

```
grids= []
```

```
_selected= None
```

```
classmethod load_grids_from_directory(grids_category_path: pathlib.Path) \[source\]
```

Loads all grids of a specific class into their class.

property **name**

This property contains a unique name for the grid.

Getter:

Gets the value of this property.

Returns:

Name of the grid as string.

Setter:

Sets the property to the provided name, if it isn't already used by another grid.

Raises:

ValueError: If the name is already used by another grid.

Type:

str

property **nodes**

This property contains all information about the nodes of a grid.

Getter:

Gets the value of this property.

Returns:

Dataframe with the information about the nodes of the grid.

Setter:

Sets the property to the provided dataframe, if the dataframe has the columns ["node","latitude","longitude"].

Type:

pd.dataframe

static **get_list_of_node_ids()** [\[source\]](#)

Return a list of all node ID numbers of the selected grid.

property **connections**

This property contains all information about the connections between nodes of a grid.

Getter:

Gets the value of this property.

Returns:

Dataframe with the information about the connections of the grid.

Setter:

Sets the property to the provided dataframe, if the dataframe has the columns ["connection","node1","node2"].

Type:

pd.dataframe

static **get_list_of_connection_ids()** [\[source\]](#)

Return a list of all node ID numbers of the selected grid.

setup_distances() [\[source\]](#)

Sets up the distances of all connections, using pint units.

Uses geodesic distance from the package GeoPy (<https://geopy.readthedocs.io/en/stable/#module-geopy.distance>) to calculate the distance between the two nodes of a connection.

```
static calculate_distances(connection) \[source\]
```

Calculates the distance or takes it from existing data.

Uses geodesic distance from the package GeoPy (<https://geopy.readthedocs.io/en/stable/#module-geopy.distance>) to calculate the distance between the two nodes of a connection, if no distance is provided in the data.

Parameters

connection – An itertuple from the connections DataFrame.

```
static selected() \[source\]
```

This methods gets the value of the static variable “selected”.

This method returns the Grid object which is selected to be used as grid for the optimization. The static variable “selected” can only contain one Grid object and is set using the method “select()” of an instance of the Grid class.

Returns

Value of _selected, which is either None or a Grid instance.

```
select() \[source\]
```

This methods sets the value of the static member “selected”.

The static member “selected” is set to reference the Grid instance which called this method.

```
class secmod.classes.Product(product_path: pathlib.Path) \[source\]
```

Bases: `object`

This class is used for all product related data, e.g. demand, impacts, etc.

```
DEFAULT_COST= 3000
```

```
products= []
```

```
classmethod load_products_from_directory(products_path: pathlib.Path) \[source\]
```

Loads all products of a specific class into their class.

```
property name
```

This property contains a unique name for the grid.

Getter:

Gets the value of this property.

Returns:

Name of the grid as string.

Setter:

Sets the property to the provided name, if it isn't already used by another grid.

Raises:

ValueError: If the name is already used by another grid.

Type:

str

```
_get_price_time_series(year: int) \[source\]
```

Returns the absolute price time series

```
_get_cost_non_served_demand(reference_year: int, invest_year: int) \[source\]
```

This method gets the costs of non-served demand of a product.

Calculates the present value of the selected impact year and its investment period in the reference year for a product.

Parameters

- **reference_year** (*int*) – Reference year of the optimized time horizon, e.g. 2020
- **invest_year** (*int*) – An investment year of the optimized time horizon, e.g. 2025

```
get_impact_non_served_demand(impact_categories: list, reference_year: int, invest_years: list)  
\[source\]
```

This methods gets the impact of non-served demand of a product in one or multiple impact categories for a specific year.

If the impact category is “cost”, the corresponding method

`Product._get_cost_non_served_demand` is used. In all other cases the impacts of all ecoinvent processes defined by the process itself as the impacts of non-served demand, are multiplied with their corresponding weight factors and summed up together.

```
static get_combined_impact_non_served_demand() \[source\]
```

Returns the combined impact time series for all products and years.

```
get_nodal_demand_time_series(year: int) \[source\]
```

Returns the absolute nodal demand time series.

```
static get_combined_demand_time_series() \[source\]
```

Returns the combined demand time series for all products and years.

```
get_required_total_secured_capacity(invest_years: list) \[source\]
```

Return the required secured capacity in the whole grid for product in a specific year.

static `get_combined_required_total_secured_capacity()` [\[source\]](#)

Returns the combined required total secured capacity for all products and years.

get_required_nodal_secured_capacity(*nodes: list, invest_years: list*) [\[source\]](#)

Return the required secured capacity in the whole grid for product in a specific year.

static `get_combined_required_nodal_secured_capacity()` [\[source\]](#)

Returns the combined required nodal secured capacity for all products and years.

classmethod `get_target_units()` [\[source\]](#)

returns dict with base units of products

class `secmod.classes.ImpactCategory`(*impact_category_path: pathlib.Path*) [\[source\]](#)

Bases: `object`

This class is used to manage impact categories, their corresponding limits and costs of overshoot.

impact_categories= []

FRAMEWORK= 'ReCiPe Midpoint (H)'

MANUAL_IMPACT_CATEGORY_SELECTION= []

property **name**

This property contains a unique name for the impact category.

Getter:

Gets the value of this property.

Returns:

Name of the impact category as string.

Setter:

Sets the property to the provided name, if it isn't already used by another impact category.

Raises:

ValueError: If the name is already used by another impact category.

Type:

str

property **ecoinvent_name**

This property contains a unique name for the impact category, which is used by ecoinvent.

Getter:

Gets the value of this property.

Returns:

ecoinvent name of the impact category as string.

Setter:

Sets the property to the provided ecoinvent name, if it isn't already used by another impact category.

Raises:

ValueError: If the ecoinvent name is already used by another impact category.

Type:

str


```
classmethod load_impact_categories_from_directory(impact_categories_path: pathlib.Path)  
[source]
```

Loads all impact categories of a specific class into their class.

```
static get_list_of_active_impact_categories() [source]
```

Returns a list of the all active impact categories in the currently selected Framework.

```
static get_list_of_names_of_active_impact_categories() [source]
```

Returns a list of the ecoinvent names of all impact categories active in the currently selected framework.

```
_get_operational_nodal_impact_limits(nodes: list, invest_years: list) [source]
```

Return the operational nodal impact limit for an impact category in a specific year.

```
static get_combined_operational_nodal_impact_limits() [source]
```

Returns the combined operational nodal impact limit for all impact categories and years.

```
_get_invest_nodal_impact_limits(nodes: list, invest_years: list) [source]
```

Return the invest nodal impact limit for an impact category in a specific year.

```
static get_combined_invest_nodal_impact_limits() [source]
```

Returns the combined invest nodal impact limit for all impact categories and years.

```
_get_total_nodal_impact_limits(nodes: list, invest_years: list) [source]
```

Return the total nodal impact limit for an impact category in a specific year.

```
static get_combined_total_nodal_impact_limits() [source]
```

Returns the combined total nodal impact limit for all impact categories and years.

`_get_operational_impact_limits(invest_years: list)` [\[source\]](#)

Return the operational impact limit for an impact category in a specific year.

`static get_combined_operational_impact_limits()` [\[source\]](#)

Returns the combined operational impact limit for all impact categories and years.

`_get_invest_impact_limits(invest_years: list)` [\[source\]](#)

Return the invest impact limit for an impact category in a specific year.

`static get_combined_invest_impact_limits()` [\[source\]](#)

Returns the combined invest impact limit for all impact categories and years.

`_get_total_impact_limits(invest_years: list)` [\[source\]](#)

Return the total impact limit for an impact category in a specific year.

`static get_combined_total_impact_limits()` [\[source\]](#)

Returns the combined total impact limit for all impact categories and years.

`_get_objective_factor_impact(invest_years: list)` [\[source\]](#)

Return the total impact limit for an impact category in a specific year.

`static get_combined_objective_factor_impact()` [\[source\]](#)

Returns the combined total impact limit for all impact categories and years.

`_get_objective_factor_impact_overshoot(invest_years: list)` [\[source\]](#)

Return the objective factors for impact overshoots for an impact category in a specific year.

`static get_combined_objective_factor_impact_overshoot()` [\[source\]](#)

Returns the combined objective factors for impact overshoots for all impact categories and years.

classmethod **get_target_units()** [\[source\]](#)

returns dict with base units of impacts

class **secmod.classes.ProcessImpacts**(*process_path: pathlib.Path = None*) [\[source\]](#)

Bases: `abc.ABC`

This class is used as an interface for process impacts.

In this class two abstract methods are defined which need to be implemented by subclasses. Therefore this class acts as an interface which is implemented by the inheriting classes. This way it is assured that the methods used to get the impacts of investment and operation can be called in all inheriting classes.

Furthermore it includes static variables for the economic time period and assumed interest rate. And since costs are special impacts, which all processes must have, the instance variables for costs are defined in this class. Furthermore the initialization of this class imports the costs from a processes directory, if provided. Methods to get costs of investment and operation are implemented as well.

Parameters

process_path (*Path*) – A path of the directory of an existing process definition, which includes a CSV-file with data about the costs of the process.

interest_rate

The interest rate used for economic impacts as share of 1.

Type

float

economic_period

The time period used for the calculation of all annualized impacts.

Type

pint quantity

Raises

FileNotFoundError – If no file “costs.csv” can be found in the process directory.

```
IMPACT_SOURCES= {None: ['operation', 'invest']}
```

```
interest_rate= 0.05
```

```
economic_period= <Quantity(30, 'year')>
```

```
invest_years= [2016, 2020, 2025, 2030, 2035, 2040, 2045, 2050]
```

```
construction_years= []
```

```
processes= []
```

property **name**

This property contains a unique name for the grid.

Getter:

Gets the value of this property.

Returns:

Name of the grid as string.

Setter:

Sets the property to the provided name, if it isn't already used by another grid.

Raises:

ValueError: If the name is already used by another grid.

Type:

str

```
abstract get_impact_invest(impact_categories: list, construction_year: int, invest_year: int) \[source\]
```

This method gets the annual investment impacts of a process

Parameters

- **impacts_categories** (*list*) – List of strings, which define for which impact categories the yearly investment impacts shall be returned
- **construction_year** (*int*) – Year of construction of an instance of this process
- **invest_year** (*int*) – Year for which the impact is calculated

```
abstract get_impact_operation(impact_categories: list, construction_year: int, reference_year: int, invest_year: int) \[source\]
```

This method gets the operational impacts of a process

Parameters

- **impacts_categories** (*list*) – List of strings, which define for which impact categories the yearly investment impacts shall be returned
- **construction_year** (*int*) – Year of construction of an instance of this process
- **invest_year** (*int*) – Year for which the impact is calculated

```
static _get_investment_period_duration(invest_year: int) \[source\]
```

Gets the duration of the period between this and the next invest year.

If it is the last year in the list of invest years, a period duration of one year is returned.

Parameters

invest_year (*int*) – A year, e.g. 2020.

_get_cost_invest(*construction_year: int, reference_year: int, invest_year: int*) [\[source\]](#)

This method gets the annual investment costs of a process

Calculates the present value of the selected impact year and its investment period in the reference year for a process instance build in the construction year.

Parameters

- **construction_year** (*int*) – Year of construction, e.g. 2016. Earliest year is used, if construction year is earlier than the earliest year in the data.
- **reference_year** (*int*) – Reference year of the optimized time horizon, e.g. 2020
- **invest_year** (*int*) – An investment year of the optimized time horizon, e.g. 2025

_get_cost_operation(*construction_year: int, reference_year: int, invest_year: int*) [\[source\]](#)

This method gets the annual investment costs of a process

Calculates the present value of the selected impact year and its investment period in the reference year for a process instance build in the construction year.

Parameters

- **construction_year** (*int*) – Year of construction, e.g. 2016. Earliest year is used, if construction year is earlier than the earliest year in the data.
- **reference_year** (*int*) – Reference year of the optimized time horizon, e.g. 2020
- **invest_year** (*int*) – An investment year of the optimized time horizon, e.g. 2025

classmethod **get_combined_impact_matrix()** [\[source\]](#)

Method that is used to return the combined impact matrix of all process of a process category.

static **_get_single_combined_impact_matrix**(combined_impact_matrix_list, process, impact_categories, construction_years, reference_year, invest_years) [\[source\]](#)

get_lifetime_duration(construction_years: list) [\[source\]](#)

Return the lifetime duration of the process for a list of construction years

classmethod **get_combined_lifetime_duration()** [\[source\]](#)

Returns the combined lifetime durations of all processes in the class using the construction years of the class.

abstract classmethod **setup_construction_years()** [\[source\]](#)

Abstract method to setup the construction years from existing capacity data.

setup_years() [\[source\]](#)

Unimplemented. See <https://git.rwth-aachen.de/ltt/secmod/secmod/issues/47>

_abc_impl= <_abc_data object>

class **secmod.classes.EcoinventImpacts**(process_path: pathlib.Path = None) [\[source\]](#)

Bases: `secmod.classes.ProcessImpacts`

This class provides impacts based on a list of ecoinvent processes.

An instance based on this class gets its impacts from the ecoinvent database. The impacts are calculated by adding a list of ecoinvent processes multiplied with a scaling factor, which are used to define the environmental impact of a process in SecMOD.

Parameters

process_path (*Path*) – A path to the directory of an ecoinvent process. If no path is provided, an empty instance will be created.

Raises

NotADirectoryError – If provided path is not a directory.

database= *None*

subassemblies= {}

```
static load_ecoinvent_database(database_path: pathlib.Path, units_to_change_path: pathlib.Path =  
None) \[source\]
```

Loads the impacts of all ecoinvent processes from a multiindexed CSV-file.

Additionally this methods loads new unit defintions to be used in pint and translates ecoinvent units to the necessary format to be used with pint.

Furthermore the process names are edited to match previous naming logic from SecMOD 1.0.

Parameters

- **database_path** (*Path*) – A path which includes the filename of the CSV-file.
- **units_to_change_path** (*Path*) – A path to a JSON-file which defines a dictionary of existing units and the units they shall be replaced with.
- **new_pint_units** (*Path*) – A path to a TXT-file which includes new unit definitions to be used as pint units. E.g. “car = []” or “mass_CO2_equivalent = [GWP100] = CO2_eq”

```
static _translate_process_name_to_ecoinvent_identifier(process_name: str) \[source\]
```

Modifies a process name to match the ecoinvent identifiers.

Deletes special characters, spaces, commas, dots, etc. and replaces them by underscores. Furthermore ensures that there is only one underscore in a row. e.g.:

“passenger car, electric, without battery//[GLO] passenger car production, electric, without battery” becomes

“passenger_car_electric_without_battery_GLO_passenger_car_production_electric_without_battery

Warning

Translating the process names from the original string to the (sadly) used style of SecMOD 1.0. This behavior should be avoided and abandoned in the future. See <https://git.rwth-aachen.de/ltt/secmod/secmod/issues/48> for more information.

```
static _translate_units_to_pint(json_path: pathlib.Path) \[source\]
```

Modifies units of impacts to match pint unit definitions.

This method simply exchanges some units from the ecoinvent database to fit the corresponding unit definitions in pint. It uses a dictionary to replace the units. This dictionary is provided as a JSON-file. The path to the file has to be given to this method or preferably to the method `load_ecoinvent_database`.

Note

If new units are needed they have to be added to this JSON-file. e.g. like this

“kg PM2.5-”: “kg PM2_5_eq”

```
static load_ecoinvent_subassemblies(subassemblies_path: pathlib.Path) \[source\]
```

Loads ecoinvent subassemblies from a directory.

This method automatically detects all CSV-files in a directory and assumes that all of them are ecoinvent subassemblies.

get_impact_invest(*impact_categories: list, construction_years: list, reference_year: int, invest_years: list*)
[\[source\]](#)

This method gets the invest impact of a process in one or multiple impact categories for a specific year.

This method determines the value of the requested impact categories for a process in a specific year. It takes into account when the process capacity was constructed, which year is the reference year of the current optimization horizon and which year is actually the current invest year.

If the impact category is “cost”, the corresponding method `ProcessImpacts.get_cost_invest` is used. In all other cases the impacts of all ecoinvent processes defined by the process itself as the invest phase, are multiplied with their corresponding weight factors and summed up together.

It returns the results as a dictionary of impact categories and their corresponding value, including the correct units.

Parameters

- **impact_categories** (*list*) – A list of one or more impact categories to be determined.
- **construction_year** (*int*) – The year in which the process capacity was build. It is used to determine the actual impact a capacity build in that year has. Earliest year is used, if construction year is earlier than the earliest year in the data.
- **reference_year** (*int*) – The reference year of the current optimization horizon. It is only used to discount the cost of the process during the whole optimization horizon to the reference year value.
- **invest_year** (*int*) – The invest year investigated right now. It is used to determine whether a process still got invest annuities to pay, only maintenance costs remaining or is already beyond its lifetime duration and therefore has no invest impacts anymore at all.

get_impact_operation(*impact_categories: list, construction_years: int, reference_year: int, invest_years: int*)
[\[source\]](#)

This method gets the operational impact of a process in one or multiple impact categories for a specific year.

This method determines the value of the requested impact categories for a process in a specific year. It takes into account when the process capacity was constructed, which year is the reference year of the current optimization horizon and which year is actually the current invest year.

If the impact category is “cost”, the corresponding method `ProcessImpacts._get_cost_operation` is used. In all other cases the impacts of all ecoinvent processes defined by the process itself as the operational phase, are multiplied with their corresponding weight factors and summed up together.

It returns the results as a DataFrame of impact categories and their corresponding value, including the correct units.

Parameters

- **impact_categories** (*list*) – A list of one or more impact categories to be determined.
- **construction_year** (*int*) – The year in which the process capacity was build. It is used to determine the actual impact a capacity build in that year has. Earliest year is used, if construction year is earlier than the earliest year in the data.
- **reference_year** (*int*) – The reference year of the current optimization horizon. It is only used to discount the cost of the process during the whole optimization horizon to the reference year value.
- **invest_year** (*int*) – The invest year investigated right now. It is only used to discount the cost of the process during the whole optimization horizon to the reference year value.

```
static _get_ecoinvent_process_impact(construction_year_attribute_name, process, impact_category:  
str, construction_year: int, impact_year: int) \[source\]
```

Get the impact of a process in a specific impact category and construction year.

Takes in an iterable tuple from a process list (e.g. `self._processes_invest`) and returns the corresponding impact value.

Parameters

- **process_list** (*pd.DataFrame*) – A DataFrame, which represents a list of processes, e.g. `self._processes_invest`
- **process** – An element of `process_list.itertuple()` which is currently evaluated.
- **impact_category** (*str*) – The name of the evaluated impact category.
- **construction_year** (*int*) – The year of construction for which the impact is determined. If the year of construction is earlier than the earliest year in the data, the earliest year is used.
- **impact_year** (*int*) – The year in which the impact actually occurs. For invest impacts the impact_year is equal to the construction_year. If the impact_year is earlier than the earliest year in the data, the earliest year is used.

Raises

KeyError – If the searched process does not exist in the ecoinvent database and seems to be used, since its weight factor is not zero.

```
static _isSubassembly(process_name: str) \[source\]
```

Checks whether a process is actually a subassembly

```
static _recursively_get_impact(process_list: pandas.core.frame.DataFrame, impact_category: str,  
construction_year: int, impact_year: int) \[source\]
```

Recursively calculates the impact of a process list

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.ManualImpact(process_path: pathlib.Path = None) \[source\]
```

Bases: `secmod.classes.ProcessImpacts`

```
get_impact_invest(impact_categories: list, construction_year: int, reference_year: int, invest_year: int)  
    \[source\]
```

```
get_impact_operation(impact_categories: list, construction_year: int, reference_year: int, invest_year: int)  
    \[source\]
```

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.Process(process_path: pathlib.Path)    \[source\]
```

Bases: `secmod.classes.ProcessImpacts`

This is the base class for processes of all kind.

It inherits from ProcessImpacts, since all processes have impacts and costs.

Parameters

process_path (*Path*) – The path to a process directory.

```
construction_years= []
```

```
locations= []
```

```
classmethod load_processes_from_directory(process_category_path: pathlib.Path)    \[source\]
```

Loads all processes of a specific class into their class.

```
get_numerical_property(property_to_return: str)    \[source\]
```

```
abstract get_existing_capacity(locations: list, invest_years: list)    \[source\]
```

```
abstract get_potential_capacity(locations: list, invest_years: list)    \[source\]
```

add_capacity(*construction_year: int, new_capacity: float*) [\[source\]](#)

Abstract method that adds capacity to a specific node or connection in a specific construction year.

classmethod **get_combined_existing_capacity**(*numtol=-1e-05*) [\[source\]](#)

numtol: float indicating the tolerance for negative values in existing capacities: small negative values are assumed to be 0

classmethod **get_combined_potential_capacity**() [\[source\]](#)

classmethod **setup_construction_years**(*combined_existing_capacity=Series([], dtype: float64)*) [\[source\]](#)

Looks up the required construction years for this process class.

A matrix of combined existing capacity can be provided to speed up the process. Otherwise the method will create a new combined_existing_capacity matrix.

classmethod **get_target_units**() [\[source\]](#)

returns dict with base units of processes

_abc_impl= <_abc_data object>

class **secmod.classes.NodalProcess**(*process_path: pathlib.Path*) [\[source\]](#)

Bases: `secmod.classes.Process`

locations= []

get_existing_capacity(*nodes: list, invest_years: list*) [\[source\]](#)

get_potential_capacity(*locations: list, invest_years: list*) [\[source\]](#)

```
get_secured_capacity_factor(construction_years: list) \[source\]
```

```
classmethod get_combined_secured_capacity_factor() \[source\]
```

Gets the secured capacity factors for all processes of a process category.

```
abstract get_technologymatrix(construction_years: list, products: list) \[source\]
```

```
classmethod get_combined_technology_matrix(products: list) \[source\]
```

```
get_maximum_production_share(nodes: list, invest_years: list) \[source\]
```

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.ConnectionProcess(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.Process`

```
locations= []
```

```
get_existing_capacity(connections: list, invest_years: list) \[source\]
```

```
abstract get_potential_capacity(locations: list, invest_years: list) \[source\]
```

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.ProductionProcess(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.NodalProcess`

```
processes= []
```

```
construction_years= []
```

```
get_technologymatrix(construction_years: list, products: list) \[source\]
```

```
get_availability_timeseries() \[source\]
```

```
static get_combined_availability_timeseries() \[source\]
```

```
get_technology_matrix_timeseries() \[source\]
```

```
static get_combined_technology_matrix_timeseries() \[source\]
```

```
static get_combined_production_products(products) \[source\]
```

Return a dictionary of all production processes and their corresponding products.

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.StorageProcess(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.NodalProcess`

```
DIRECTIONS_STORAGE= {None: ['deposit', 'withdraw']}
```

```
STORAGE_LEVEL_FACTOR= {'deposit': 1, 'withdraw': -1}
```

```
processes= []
```

```
_abc_impl= <_abc_data object>
```

```
construction_years= []
```



```
get_technologymatrix(construction_years: list, products: list) \[source\]
```

```
get_flow_to_storage_capacity_factor(construction_years: list) \[source\]
```

```
static get_combined_flow_to_storage_capacity_factor() \[source\]
```

```
static get_combined_storage_products(products) \[source\]
```

Return a dictionary of all storage processes and their storeable product.

```
class secmod.classes.TransshipmentProcess(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.ConnectionProcess`

```
_abc_impl= <_abc_data object>
```

```
DIRECTIONS_TRANSSHIPMENT= {None: ['forward', 'backward']}
```

```
DIRECTION_FACTOR_TRANSSHIPMENT= {'backward': -1, 'forward': 1}
```

```
processes= []
```

```
construction_years= []
```

```
get_transshipment_efficiency() \[source\]
```

Return a dataframe with the efficiency of every connection for a transshipment process.

```
get_potential_capacity(locations: list, invest_years: list) \[source\]
```

```
classmethod get_combined_transshipment_efficiency() \[source\]
```

Return a dataframe with the efficiencies of every connection for every transshipment process.

```
classmethod get_combined_transshipment_products() \[source\]
```

Returns a dictionary with the names and products of all transshipment processes

```
class secmod.classes.TransmissionProcess(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.ConnectionProcess`

```
_abc_impl = <_abc_data object>
```

```
PRODUCTS = {None: ['electricity']}
```

```
processes = []
```

```
construction_years = []
```

```
per_unit_base = 500.0
```

```
reference_node = 1
```

```
get_potential_capacity(locations: list, invest_years: list) \[source\]
```

```
_get_process_connection_properties(connections: list, invest_years: list) \[source\]
```

```
static _get_process_by_voltage(voltage: <module 'pint.quantity' from  
'C:\\Users\\glrunner\\AppData\\Roaming\\Python\\Python37\\site-packages\\pint\\quantity.py'>)  
\[source\]
```

```
static get_connection_properties() \[source\]
```

```
static get_combined_power_limit_per_circuit() \[source\]
```

Returns a dictionary of the power limits per circuit of all transmission processes.

```
static get_combined_safety_margin() \[source\]
```

Returns a dictionary of the safety margin of all transmission processes.

```
class secmod.classes.ProductionProcessEcoinvent(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.ProductionProcess`, `secmod.classes.EcoinventImpacts`

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.ProductionProcessManual(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.ProductionProcess`, `secmod.classes.ManualImpact`

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.StorageProcessEcoinvent(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.StorageProcess`, `secmod.classes.EcoinventImpacts`

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.StorageProcessManual(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.StorageProcess`, `secmod.classes.ManualImpact`

```
_abc_impl= <_abc_data object>
```

```
class secmod.classes.TransshipmentProcessEcoinvent(process_path: pathlib.Path) \[source\]
```

Bases: `secmod.classes.TransshipmentProcess`, `secmod.classes.EcoinventImpacts`

```
_abc_impl= <_abc_data object>
```

class `secmod.classes.TransshipmentProcessManual`(*process_path: pathlib.Path*) [\[source\]](#)

Bases: `secmod.classes.TransshipmentProcess`, `secmod.classes.ManualImpact`

```
_abc_impl= <_abc_data object>
```

class `secmod.classes.TransmissionProcessEcoinvent`(*process_path: pathlib.Path*) [\[source\]](#)

Bases: `secmod.classes.TransmissionProcess`, `secmod.classes.EcoinventImpacts`

```
_abc_impl= <_abc_data object>
```

class `secmod.classes.TransmissionProcessManual`(*process_path: pathlib.Path*) [\[source\]](#)

Bases: `secmod.classes.TransmissionProcess`, `secmod.classes.ManualImpact`

```
_abc_impl= <_abc_data object>
```

`secmod.classes.calculate_power_line_power_limit`(*power_limit, safety_margin, circuits*)
[\[source\]](#)

`secmod.classes.calculate_power_line_resistance_per_unit`(*specific_resistance, distance, circuits, voltage, per_unit_base*) [\[source\]](#)

`secmod.classes.calculate_power_line_susceptance_per_unit`(*reactance_per_unit, resistance_per_unit*) [\[source\]](#)

`secmod.classes.calculate_voltage_switch_power_limit`(*target_power_limit, base_power_limit, safety_margin, circuits*) [\[source\]](#)

`secmod.classes.calculate_voltage_switch_resistance_per_unit`(*target_voltage, base_voltage, target_specific_resistance, base_specific_resistance, per_unit_base, circuits, distance*) [\[source\]](#)

`secmod.classes.get_target_unit_from_unit`(*unit: str*) [\[source\]](#)

convert Unit String to Target Unit and returns target unit string

Parameters

unit (*str*) – string of unit

`secmod.classes.convert_quantity_to_correct_unit(unit_input)` [\[source\]](#)

compare unit from csv with unit in target unit dictionary

$m \cdot d' = m_{\text{target}} \cdot d_{\text{target}} \leftrightarrow m_{\text{target}} = m \cdot d' / d_{\text{target}}$ with m: magnitude, d: dimensionality

Parameters

unit_input – unit_input which oughts to be converted into correct unit. currently supported for pd.DataFrame, pd.Series and str

Returns

unit_input converted into correct unit.

Return type

converted_unit_input

`secmod.classes.calculate_multiplication_factors(unit, multiplication_factors=None)` [\[source\]](#)

returns multiplication factor of conversion from current unit to target unit

Parameters

- **unit** – current unit or quantity
- **multiplication_factors** (*list*) – list with multiplication factors, to which the current factor is added

secmod.data_preprocessing module

`secmod.data_preprocessing.download_datapackage(url: str, download_path: pathlib.Path) → bool`

[source]

This method is used to download an Open Source Data Package.

Using the URL to the meta data JSON-file of the data package, the complete data package is downloaded into a download folder. If the download has been successful a boolean True is returned.

Parameters

- **url** (*str*) – URL to the directory of the meta data JSON-file of the data package (e.g. https://data.open-power-system-data.org/renewable_power_plants/latest/datapackage.json)
- **download_path** (*Path*) – Path of the directory where the data package is to be saved

```
secmod.data_preprocessing.datapackage_update_available(url: str, download_path:  
pathlib.Path) → bool [source]
```

This method checks if there is an update available for a remote data package.

First the method tries to open the remote data package to check its availability. If it is not available, False is returned.

After that the local data package is opened, if it exists. If it does not exist, True is returned.

If both sources are accessible the version information from the meta data is compared. Because the remote source is expected to have the newest version, any difference in the version information returns True.

Parameters

- **url** (*str*) – URL to the directory of the meta data JSON-file of the data package (e.g. https://data.open-power-system-data.org/renewable_power_plants/latest/datapackage.json)
- **download_path** (*Path*) – Path of the directory where the data package is to be saved

```
secmod.data_preprocessing.download_file(url: str, save_to: str, expected_size: int = None) [source]
```

This method downloads files using streaming.

A get request is used to download the file from the URL. The response is streamed into a local file to minimize RAM usage.

Parameters

- **url** (*str*) – URL to the file
- **save_to** (*str*) – local path and filename to be saved to

```
secmod.data_preprocessing.is_setup(working_directory: pathlib.Path) → bool \[source\]
```

This method checks if SecMOD has been set up before.

This is achieved by checking if the folder “SecMOD” exists in the working directory and returning a boolean. If it does not exist, the method returns False.

Parameters

working_directory (*Path*) – Path of the working directory

```
secmod.data_preprocessing.create_directory(path: pathlib.Path) \[source\]
```

This method creates a new directory, if it does not exist, yet.

The method uses pathlib to check for the existence of a path. If the path does not exist yet, it get created.

Parameters

path (*Path*) – Path of the directory to be created

secmod.data_processing module

```
secmod.data_processing.load_raw_input(raw_input_directory: pathlib.Path) \[source\]
```

Loads all data from raw input.

```
secmod.data_processing.load_all_processes(processes_directory: pathlib.Path) \[source\]
```

Loads all processes in their corresponding class.

```
secmod.data_processing.generate_input_dictionary(computed_input_path: pathlib.Path, load: bool = False) \[source\]
```

Takes all loaded data and generates an input dictionary for the optimization model.

```
secmod.data_processing.get_dimensionalities_from_input_dictionary(input_dictionary: dict) \[source\]
```

Returns a list with all unique dimensionalities and an example unit

```
secmod.data_processing.add_new_capacity_from_results(optimization_model: secmod.optimization.Optimization, years: list = None) \[source\]
```

Add the returned new capacity from the optimization to the existing capacity in the data

secmod.evaluation module

```
class secmod.evaluation.AbstractFrame_evaluation(parent, optimization_results, working_directory, frac_height=0.8, frac_width=0.8) \[source\]
```

Bases: `tkinter.Frame`

This is the abstract GUI Frame class

It is used to define the common parts of the MainFrame and the DetailedFrame.

```
startPlot() \[source\]
```

start setup or plot of figure

```
setupFigure() \[source\]
```

set up figure in north frame

calculateFrameGeometry(*idxrow=0, idxcolumn=0*) [\[source\]](#)

Calculate width and height of frame in GUI

plotFigure() [\[source\]](#)

plot balance

getbalanceUnits(*grouped_balance, unit_time_summation, b_extendedDataframe=False*) [\[source\]](#)

Get units of balance in optimization

selectPlotType() [\[source\]](#)

plots the data according to specified plot type

createLegend() [\[source\]](#)

creates seperate Legends

setColors() [\[source\]](#)

return list of colors for plot

highlightMouseSelection(*event*) [\[source\]](#)

highlight selection over which mouse currently hovers

showTable(*extendedDataframe=False*) [\[source\]](#)

Show table with values of dataframe

adjustManualColumnWidths() [\[source\]](#)

Adjust column width of table. Copied from pandastable documentation and cleared line "if w > 200"

savePlotData() [\[source\]](#)

opens new Save Frame

```
groupBalanceByIndex(input_balance) \[source\]
```

in mainframe no grouping necessary. Method overwritten in Detailed Frame

```
getNonzeroIndizes(summed_level=None) \[source\]
```

get indizes of balance whose values are nonzero

```
abstract getTitleString() \[source\]
```

abstract method. Needs to be implemented in Child Class

```
class secmod.evaluation.MainFrame_evaluation(parent, optimization_results, working_directory)  
\[source\]
```

Bases: `secmod.evaluation.AbstractFrame_evaluation`

This is the Main Frame of the evaluation GUI

It is a child class of AbstractFrame_evaluation

```
setUpRadiobuttons(*args) \[source\]
```

Set up and change radiobuttons for product flows/impacts/processes

```
getDataframeFromSelection(*args) \[source\]
```

Create Dataframe which contains the information representing the selection, i.e. selection of Product, Balance and Impact/Capacities

```
disableEnableRadiobuttons() \[source\]
```

Disable radiobuttons of balance specification, which is not available for product

Only necessary for single product

plotFigure() [\[source\]](#)

plot balance

inherited from AbstractFrame_evaluation but click functionality added

openDetailedWindow(event) [\[source\]](#)

open Detailed Frame to further inspect line/bar/area clicked on

getTitleString() [\[source\]](#)

Get Title String depended on selection of balance

```
class secmod.evaluation.DetailedFrame_evaluation(parent, optimization_results, working_directory,  
reduced_balance, mainframe) \[source\]
```

Bases: `secmod.evaluation.AbstractFrame_evaluation`

This is the Detailed Frame of the evaluation GUI

It is a child class of AbstractFrame_evaluation

disableEnableCheckbuttons() [\[source\]](#)

Disable checkbuttons of aggregation level, which is not available in balance

groupBalanceByIndex(input_balance) [\[source\]](#)

Group balance by level of aggregation. Index selected by checkbuttons

showTable() [\[source\]](#)

show table

select whether detailed or entire dataset shown in table

getTitleString() [\[source\]](#)

Get Title String depended on selection of balance

Same Title as MainFrame, only 'Detailed' added

```
class secmod.evaluation.SaveFrame(parent, mainframe) \[source\]
```

Bases: `object`

New Frame in order to save plot and Data

```
toggleSaveButton() \[source\]
```

disables the SaveButton if no checkbutton selected

```
browseFolder() \[source\]
```

Browses Folder and selects new Folder

```
saveSelection() \[source\]
```

saves the selection as <filename> in <folderpath>

```
secmod.evaluation.evaluate(working_directory) \[source\]
```

Evaluates the results of the optimization.

```
secmod.evaluation.start_evaluation(working_directory, debug_evaluation=False) \[source\]
```

secmod.helpers module

```
secmod.helpers.isInteger(text: str) \[source\]
```

Checks if string could be an integer.

```
secmod.helpers.unitize_dataframe(dataframe: pandas.core.frame.DataFrame, units:  
pint.registry.UnitRegistry) \[source\]
```

Multiplied every row of a dataframe with the corresponding unit and drops the unit column.

`secmod.helpers.log_heading(message: str)` [\[source\]](#)

Takes a message and creates a nicely formatted heading log message.

`secmod.helpers.clear_filename_special_character(filename: str)` [\[source\]](#)

`secmod.helpers.convert_date(date_old, correct_dateformat, partially_correct_dateformat, wrong_dateformat)` [\[source\]](#)

converts date to correct dateformat. The old date must be of a known dateformat

Parameters

- `date_old` (*str*) – String of date in old format
- `correct_dateformat` (*str*) – String of known dateformat, to which the date is converted
- `partially_correct_dateformat` (*str*) – String of known, wrong dateformat, which is similar to `correct_dateformat` (though it actually does not matter how similar it is)
- `wrong_dateformat` (*str*) – String of known, wrong dateformat

`secmod.helpers.correct_time_stamp_of_timeseries(timeseries)` [\[source\]](#)

corrects time stamp of timeseries

format desired, which is given in config (e.g. dd.mm.yyyy HH:MM)

Parameters

`timeseries` (*pd.MultiIndex*) – timeseries with dates

`secmod.helpers.get_rwth_colors()` [\[source\]](#)

return RWTH colors as array

`secmod.helpers.shorten_ecoinvent()` [\[source\]](#)

secmod.optimization module

class **secmod.optimization.Optimization** [\[source\]](#)

Bases: `abc.ABC`

This is the class for optimization models.

It includes all declarations of Sets, Parameters, Variables and Constraints. Furthermore it contains all optimization methods.

```
variable_dict= {}
```

```
num_constraints= {}
```

```
numerics= {}
```

static **calculate_scaling_factor**(*scaling_vector*) [\[source\]](#)

returns scaling factor r or c for scaling_vector as Arg and scaling algorithm selected in config
:param scaling_vector: list with coefficients :type scaling_vector: list

Returns

scaling factor corresponding to scaling_vector

Return type

scaling_factor(float)

classmethod **set_coefficients_for_numerics**(*item, scaled_model: bool, constraint_name='<constraint unknown>'*) [\[source\]](#)

this classmethod receives an item containing a tuple of a variable and its corresponding coefficient. It subsequently compares the coefficient to the saved max and min value and saves the coefficient if it is larger/smaller. Furthermore it sums up the coefficients in order to calculate the frobenius norm.

Parameters

- **cls** – class Optimization
- **item** (*tuple*) – tuple which contains coefficient [0] and variable [1]
- **scaled_model** (*bool*) – boolean whether scaled or unscaled model
- **constraint_name** (*str*) – str which contains name of constraint. During the construction of the constraint, it is impossible and impractical to pass the constraint name. If constraint of min/max coefficient can not be derived from context, disable scaling

classmethod **calculate_matrix_properties()** [\[source\]](#)

this classmethod calculates the highest and lowest matrix coefficient, and the frobenius norm for the scaled and unscaled model Performed after initialization of model. :param cls: class Optimization

classmethod **set_variable_dict**(*variable, variable_coefficient, constraint_scaling_factor*) [\[source\]](#)

appends variable_coefficient to variable_dict :param cls: class Optimization :param variable: indexed variable :type variable: pe.Var :param variable_coefficient: value of coefficient corresponding to variable :type variable_coefficient: float :param constraint_scaling_factor: scaling factor of constraint :type constraint_scaling_factor: float

classmethod **add_evaluated_constraint()** [\[source\]](#)

this method increases the counter of evaluated constraints by one

classmethod **add_skipped_constraint()** [\[source\]](#)

this method increases the counter of skipped constraints by one

classmethod **scale_expr**(*unscaled_expr*) [\[source\]](#)

this method scales an constraint expression, if `config.scaling_options['scale_constraints']` or `config.scaling_options['scale_variables']` selected. If `config.scaling_options['scale_constraints']` the constraint is scaled. If `config.scaling_options['scale_variables']` selected, this methods saves the variable coefficients in `cls.variable_dict`

Parameters

- **cls** – class Optimization
- **unscaled_expr** (*EXPR.InequalityExpression/EXPR.EqualityExpression*) – unscaled inequality or equality expression, which is generated by the calling constraint

Returns

unscaled inequality or equality expression,
used in the calling constraint

Return type

scaled_expr(*EXPR.InequalityExpression/EXPR.EqualityExpression*)

static fix_variables(*args) [\[source\]](#)

This method sets the variables that need to be fixed to 0 at initialization

By setting the default value to 0, the fixed variables can be identified before explicitly fixing them. If condition is met, function returns 0, otherwise None. This is subsequently set as the variable's default value :param *args: args[0](str): string of variable name to identify variable

args[1](*pe.ConcreteModel*): *ConcreteModel* with necessary sets and parameters args[2:]: Set indices which define the current variable index (e.g. year, product, production_process, node etc.)

setupSets() [\[source\]](#)

This method sets up all Sets required the optimization model.

Some Sets are initialized with default values, if no other values are specified in the input file.

setupParameters() [\[source\]](#)

This method sets up all Parameters required by the optimization model.

setupScalingParameters() [\[source\]](#)

This functions sets up the scaling parameters of the variables required by the optimization model.

The notation is the following: The scaling parameter for the component <component> is self.model.scaling_<component> It is crucial to select the same pe.Sets as the corresponding component, to select default = 1, and mutable=True. Check setupVariables() for details on the variable declaration

setupVariables() [\[source\]](#)

This method sets up all Variables required by the optimization model.

setupConstraints() [\[source\]](#)

This method sets up all Constraints required by the optimization model.

setupObjective() [\[source\]](#)

This method declares the pe.Objective required by the optimization model.

scale_model_variables() [\[source\]](#)

scales model variables, which are in Optimization.variable_dict

retrieve_small_bounds_and_calculate_numerics() [\[source\]](#)

retrieves and prints bound that are smaller than eps_bound. Furthermore set coefficients for numerics of scaled model and calculate matrix properties

```
instantiate_model(input_dict: dict = None, filepath: str = None, skip_instantiation: bool = False)  
[source]
```

This method creates a model instance, fixes variables, and, if selected, analyzes numerics

Parameters

- **input_dict** (*dict*) – input dictionary
- **filepath** (*str*) – filepath to input dictionary
- **skip_instantiation** (*bool*) – bool if instantiation is skipped. Default is False

```
run(input_dict: dict = None, filepath: str = None, solver: str = 'glpk', solver_options: dict = None, debug: bool =  
False, skip_instantiation: bool = False) [source]
```

This method is used to start an optimization.

To start an optimization it is necessary to provide a file with the necessary input data. This file includes all information required to instantiate the optimization problem.

Parameters

- **input_dict** (*dict*) – Input dictionary that contains all the necessary data for the optimization
- **filename** (*str*) – Filename of the previously generated input file
- **solve** (*str*) – Name of the solve to be used. Standard is 'gurobi'
- **solver_options** (*dict*) – dict with options directly forwarded to the solver
- **debug** (*bool*) – bool if solver is debugged. Default is False
- **skip_instantiation** (*bool*) – bool if instantiation is skipped. Default is False

```
fix_unneeded_variables() [source]
```

Fixes variables to 0. The variables have to be labeled as to be fixed in self.fix_variables
Here, we iterate through each variable and check if the default value is 0. If yes, the variable is fixed

fix_slack_variables(*fix_products*) [\[source\]](#)

Fixes the non_served_demand of specified products(e.g. heat, electricity, and mobility) to 0. Although the import of these products may be undesired in some models, their import is usually allowed at arbitrarily high environmental or financial impacts to keep the optimization feasible. By fixing the non-served-demand to 0, these variables are no longer used as slack variables and their import is strictly prevented

get_new_capacity_from_result() [\[source\]](#)

Return a dictionary of dataframes with all new capacities from the optimization.

static **get_dataframe_from_result**(*model_instance_variable, variable_scaling_factors, unstack: list = None*) [\[source\]](#)

Returns a dataframe with the results from the optimization model.

static **get_dataframe_from_parameter**(*model_instance_variable, unstack: list = None*) [\[source\]](#)

Returns a dataframe with the results from the optimization model.

static **constraint_product_balance_rule**(*model, node: int, product: str, year: int, time_slice: str*) → pyomo rule [\[source\]](#)

This method is used as rule for the product balance equation.

This expression represents the product balance equation which is unique for each node, product, year and time slice. It can be summarized as:

$$\text{Production} + \text{Transshipment} + \text{Storage} + \text{Transmission} \geq \text{Demand}$$

Therefore it takes the named above as parameters.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active

- **product** (*str*) – A product at which is constrained
- **year** (*int*) – A year at which the pe.Constraint is active
- **time_slice** (*str*) – A time_slice at which the pe.Constraint is active

```
static constraint_production_limit_by_capacity_rule(model, node: int, process_production: str, year: int, year_construction: int, time_slice: str) → pyomo rule \[source\]
```

This method is used as rule for limiting the production to the sum of existing and new capacity.

The rule is applied to every year of operation AND every year of construction. This means that for every year the capacity of existing infrastructure is used as limit, creating a constraint for every year of construction as well. This way the operation can differentiate between older and newer capacities which might have different impacts. Additionally for the years of construction which also are investment years, the newly build capacities are added to the limit.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active
- **process_production** (*str*) – A production process which is constrained
- **year** (*int*) – A year at which the pe.Constraint is active
- **year_construction** (*int*) – The year of construction of a specific capacity
- **time_slice** (*str*) – A time_slice at which the pe.Constraint is active

```
static constraint_transshipment_limit_by_capacity_rule(model, connection: int, process_transshipment: str, direction_transshipment: str, year: int, year_construction: int, time_slice: str) → pyomo rule \[source\]
```

This method is used rule for limiting the used transmission up to the sum of existing and new capacity.

The rule is applied to every year of operation AND every year of construction. This means that for every year the capacity of existing infrastructure is used as limit, creating a constraint for every year of construction as well. This way the operation can differentiate between older and newer capacities which might have different impacts. Additionally for the years of construction which also are investment years, the newly build capacities are added to the limit.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **process_transshipment** (*str*) – A transshipment process which is constrained
- **direction_transshipment** (*str*) – A transshipment direction which is constrained
- **year** (*int*) – A year at which the pe.Constraint is active
- **year_construction** (*int*) – The year of construction of a specific capacity
- **time_slice** (*str*) – A time_slice at which the pe.Constraint is active

static **constraint_storage_level_limit_by_capacity_rule**(*model*, *node*: *int*, *process_storage*: *str*, *year*: *int*, *year_construction*: *int*, *time_slice*: *str*) → pyomo rule [\[source\]](#)

This method is used as rule for limiting the used storage up to the sum of existing and new capacity.

The rule is applied to every year of operation AND every year of construction. This means that for every year the capacity of existing infrastructure is used as limit, creating a constraint for every year of construction as well. This way the operation can differentiate between older and newer capacities which might have different impacts. Additionally for the years of construction which also are investment years, the newly build capacities are added to the limit.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active

- **process_storage** (*str*) – A storage process which is constrained
- **year** (*int*) – A year at which the pe.Constraint is active
- **year_construction** (*int*) – The year of construction of a specific capacity
- **time_slice** (*str*) – A time_slice at which the pe.Constraint is active

static constraint_used_storage_limit_by_capacity_rule(*model, node: int, process_storage: str, direction_storage: str, year: int, year_construction: int, time_slice: str*) → pyomo rule [\[source\]](#)

This abstract method is used as rule for limiting the output and input of storage processes.

The maximum flow out of or into a storage instance is limited by the available storage capacity multiplied with a flow-to-storage-capacity factor.

Although the available capacity can be defined differently, e.g.:

```
storage usage <= flow-to-storage-capacity factor * (installed capacity)
```

or:

```
storage usage <= flow-to-storage-capacity factor * (installed capacity + new capacity)
```

Because of this, the abstract pe.Constraint rule needs to be defined in a subclass.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active
- **process_storage** (*str*) – A storage process which is constrained
- **direction_storage** (*str*) – A direction of “withdraw” or “deposit” a product
- **year** (*int*) – A year at which the pe.Constraint is active
- **year_construction** (*int*) – The year of construction of a specific capacity

- **time_slice** (*str*) – A time_slice at which the pe.Constraint is active

```
static constraint_storage_level_rule(model, node: int, process_storage: str, year: int,  
year_construction: int, time_slice: str) → pyomo rule \[source\]
```

This method is used as rule for the calculation of the storage level.

Since the storage should not act as a source or sink, this method calculates the storage level after each time_slice and couples it with the storage level at the beginning of the next time slice.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active
- **process_storage** (*str*) – A storage process which is constrained
- **year** (*int*) – A year at which the pe.Constraint is active
- **year_construction** (*int*) – The year of construction of a specific capacity
- **time_slice** (*str*) – A time_slice at which the pe.Constraint is active

```
static constraint_transmission_limit_by_capacity_positive_rule(model, connection: int,  
year: int, time_slice: str) → pyomo rule \[source\]
```

This method is used as rule for limiting the load flow through a transmission power line.

The load flow needs to be limited by the available capacity, which is why this abstract rule exists and is used for a constraint in the base class for all optimizations. Although the available capacity can be defined differently, e.g.:

```
load flow <= installed capacity
```

or:

```
load flow <= installed capacity + capacity from switching + capacity from new power line
```

Because of this, the abstract `pe.Constraint` rule needs to be defined in a subclass.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **connection** (*int*) – A power line which is constrained
- **year** (*int*) – A year at which the `pe.Constraint` is active
- **time_slice** (*str*) – A `time_slice` at which the `pe.Constraint` is active

```
static constraint_transmission_limit_by_capacity_negative_rule(model, connection: int,  
year: int, time_slice: str) → pyomo rule \[source\]
```

This method is used as rule for limiting the load flow through a transmission power line.

The load flow needs to be limited by the available capacity, which is why this abstract rule exists and is used for a constraint in the base class for all optimizations. Although the available capacity can be defined differently, e.g.:

```
load flow <= installed capacity
```

or:

```
load flow <= installed capacity + capacity from switching + capacity from new power line
```

Because of this, the abstract `pe.Constraint` rule needs to be defined in a subclass.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **power_line** (*int*) – A power line which is constrained
- **year** (*int*) – A year at which the `pe.Constraint` is active

- **time_slice** (*str*) – A time_slice at which the pe.Constraint is active

```
static constraint_operational_nodal_impact_rule(model, node: int, impact_category: str, year: int) → pyomo rule \[source\]
```

This method is used as rule for the calculation of the operational impact vector summed over all processes.

The operational impact of one node over all time slices is calculated in this method. By summing over the construction years of all capacities, the corresponding impact matrices from the actual construction year can be used. Furthermore the impact calculated for each time slice is multiplied with the weight in hours of a year the time slice has.

The operational impact of production processes equals the sum of the products of process usage and the corresponding impact matrix. The same method is used for calculating the impact of transshipment processes, but the resulting impact is shared equally between the corresponding nodes.

The operational impact of storage processes is calculated based on the **withdrawn** product flow.

For the electric transmission grid it is assumed that the operational impact is negligible.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

```
static constraint_operational_nodal_impact_limits_rule(model, node: int, impact_category: str, year: int) → pyomo rule \[source\]
```

This method is used as rule for the limitation of the operational impact vector summed over all processes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

```
static constraint_operational_impact_rule(model, impact_category: str, year: int) → pyomo rule  
\[source\]
```

This method is used as rule for the calculation of the operational impact vector summed over all processes and nodes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

```
static constraint_operational_impact_limits_rule(model, impact_category: str, year: int) →  
pyomo rule \[source\]
```

This method is used as rule for the limitation of the operational impact vector summed over all processes and nodes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

```
static constraint_invest_nodal_impact_rule(model, node: int, impact_category: str, year: int) →  
pyomo rule \[source\]
```

This method is used as rule for the calculation of the invest impact vector summed over all processes.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

```
static constraint_invest_nodal_impact_limits_rule(model, node: int, impact_category: str,  
year: int) → pyomo rule \[source\]
```

This method is used as rule for the limitation of the invest impact vector summed over all processes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

```
static constraint_invest_impact_rule(model, impact_category: str, year: int) → pyomo rule  
\[source\]
```

This method is used as rule for the calculation of the invest impact vector summed over all processes and nodes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

static **constraint_invest_impact_limits_rule**(*model*, *impact_category*: str, *year*: int) → pyomo rule [\[source\]](#)

This method is used as rule for the limitation of the invest impact vector summed over all processes and nodes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

static **constraint_total_nodal_impact_rule**(*model*, *node*: int, *impact_category*: str, *year*: int) → pyomo rule [\[source\]](#)

This method is used as rule for the calculation of the total impact vector summed over all processes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

static **constraint_total_nodal_impact_limits_rule**(*model*, *node*: int, *impact_category*: str, *year*: int) → pyomo rule [\[source\]](#)

This method is used as rule for the limitation of the total impact vector summed over all processes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node** (*int*) – A node at which the pe.Constraint is active

- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

static **constraint_total_impact_rule**(*model, impact_category: str, year: int*) → pyomo rule
[\[source\]](#)

This method is used as rule for the calculation of the total impact vector summed over all processes and nodes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

static **constraint_total_impact_limits_rule**(*model, impact_category: str, year: int*) → pyomo rule
[\[source\]](#)

This method is used as rule for the limitation of the total impact vector summed over all processes and nodes

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **impact_category** (*str*) – An impact category which is assessed
- **year** (*int*) – A year at which the pe.Constraint is active

static **constraint_production_potential_capacity_by_input_rule**(*model, node: int, process_production: str, year: int*) → pyomo rule
[\[source\]](#)

This method is used as rule for limiting the capacity of production processes by limits from the input e.g. maximum potential.

Some production technologies like renewable energies have local capacity limits based on the availability of resources or regulatory restrictions. These limits are given to the

optimization as parameter and can change by time.

Args;

model: The equivalent of “self” in pyomo optimization models
node (int): A node at which the pe.Constraint is active
process_production (str): A production process which is constrained
year (int): A year at which the pe.Constraint is active

```
static constraint_storage_potential_capacity_by_input_rule(model, node: int,  
process_storage: str, year: int) → pyomo rule \[source\]
```

This method is used as rule for limiting the capacity of storage processes by limits from the input e.g. maximum potential

Some storage technologies like gas cavern storages have local capacity limits based on the availability of resources or regulatory restrictions. These limits are given to the optimization as parameter and can change by time.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **node (int)** – A node at which the pe.Constraint is active
- **process_storage (str)** – A storage process which is constrained
- **year (int)** – A year at which the pe.Constraint is active

```
static constraint_transshipment_potential_capacity_by_input_rule(model, connection:  
int, process_transshipment: str, year: int) → pyomo rule \[source\]
```

This method is used as rule for limiting the capacity of transshipment processes by limits from the input e.g. maximum potential.

Some transshipment technologies like pipelines have local capacity limits based on the availability of resources or regulatory restrictions. These limits are given to the optimization as parameter and can change by time.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **connection** (*int*) – A connection of the grid
- **process_transshipment** (*str*) – A transshipment process which is constrained
- **year** (*int*) – A year at which the pe.Constraint is active

static constraint_transmission_potential_capacity_by_input_rule(*model*, *connection: int*, *process_transmission: str*, *year: int*) → pyomo rule [\[source\]](#)

This method is used as rule for limiting the capacity of transmission processes by limits from the input e.g. maximum potential.

Some transmission technologies have local capacity limits based on the availability of resources or regulatory restrictions. These limits are given to the optimization as parameter and can change by time.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **connection** (*int*) – A connection of the grid
- **process_transmission** (*str*) – A transmission process which is constrained
- **year** (*int*) – A year at which the pe.Constraint is active

static constraint_total_secured_capacity_rule(*model*, *product: str*, *year: int*) → pyomo rule [\[source\]](#)

This method is used as rule for providing the necessary secured capacity for a product.

A baseline of required secured capacity can be given as parameter. This required secured capacity needs to be lower or equal to the sum of capacity secured by existing and new capacities.

Since some technologies might increase the need for secured capacity for a product, their impact is taken into account as well.

Only production processes are taken into account for the secured capacity calculation.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **product** (*str*) – A product for which the pe.Constraint is active
- **year** (*int*) – A year at which the pe.Constraint is active

```
static constraint_nodal_secured_capacity_rule(model, node: int, product: str, year: int) →  
pyomo rule \[source\]
```

This method is used as rule for providing the necessary secured capacity for a product.

A baseline of required secured capacity can be given as parameter. This required secured capacity needs to be lower or equal to the sum of capacity secured by existing and new capacities.

Since some technologies might increase the need for secured capacity for a product, their impact is taken into account as well.

Only production processes are taken into account for the secured capacity calculation.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models
- **product** (*str*) – A product for which the pe.Constraint is active
- **year** (*int*) – A year at which the pe.Constraint is active

```
static objectiveRule(model) \[source\]
```

This method is used as an pe.Objective for the optimization, using weight factors for all impact categories.

Parameters

- **model** – The equivalent of “self” in pyomo optimization models


```
_abc_impl= <_abc_data object>
```

secmod.setup module

```
secmod.setup.setup(working_directory: pathlib.Path, reset: bool = False, download_ext_datapackages: bool = False) \[source\]
```

This methods sets up SecMOD for the first use in the working directory.

This method copies the embedded sample data to the working directory. Furthermore it downloads the external data packages for the first time. Last but not least it creates a python file and a batch file to easily start SecMOD.

Parameters

- **working_directory** (*Path*) – The folder which is used as working directory by SecMOD
- **reset** (*bool*) – Boolean value which decides whether an existing working directory is overwritten
- **download_external_datapackages** (*bool*) – Boolean value which decides whether external datapackages are downloaded

```
secmod.setup.copy_sample_data(working_directory: pathlib.Path) \[source\]
```

This method copies sample data from the package to the working directory.

Parameters

- **working_directory** (*Path*) – The folder which is used as working directory by SecMOD

```
secmod.setup.create_startup_helper(working_directory: pathlib.Path) \[source\]
```

This method creates startup helpers in the working directory.

The startup helpers created by this method allow to run SecMOD with one click.

Parameters

working_directory (*Path*) – The folder which is used as working directory by SecMOD

```
secmod.setup.download_external_datapackages(working_directory: pathlib.Path)
```

[\[source\]](#)

Module contents

[↩ Previous](#)[Next ➞](#)

© Copyright 2021, Institute of Technical Thermodynamics, RWTH Aachen University

Built with [Sphinx](#) using a [theme](#) provided by [Read the Docs](#).

Packages in SecMOD

Package name	Usage	Link
Pyomo	Optimization	https://www.pyomo.org/
Numpy	Scientific Computing	https://numpy.org/
Pandas	Scientific Computing	https://pandas.pydata.org/
Scipy	Scientific Computing	https://www.scipy.org/
datapackage	Datamanagement	https://frictionlessdata.io/docs/using-data-pac
geopy	Geocoordinates/distances	https://github.com/geopy/geopy
pint	Unit-Handling	https://pint.readthedocs.io/en/latest/
pathlib		https://docs.python.org/3/library/pathlib.html
requests	Network	https://3.python-requests.org/
clint	Terminal-Visualation	https://github.com/kennethreitz/clint
tikzplotlib	Visualization	https://github.com/nschloe/tikzplotlib
sympy		https://www.sympy.org/en/index.html
tsam	Scientific Computing	https://github.com/FZJ-IEK3-VSA/tsam
pandastable	Visualization	https://pandastable.readthedocs.io/en/latest/d

Troubleshooting

SecMOD is programmed to handle most things internally, if possible. If you still encounter an error, SecMOD mostly provides you with an error message which should help to resolve the issue.

[← Previous](#)

© Copyright 2021, Institute of Technical Thermodynamics, RWTH Aachen University

Built with [Sphinx](#) using a [theme](#) provided by [Read the Docs](#).