OMEGAlpes Documentation

Release 0.0.1

See OMEGAlpes authors

Jan 20, 2023

Contents

CHAPTER 1

Introduction to OMEGAlpes

OMEGAlpes stands for Generation of Optimization Models As Linear Programming for Energy Systems. It aims to be an energy systems modelling tool for linear optimisation (LP, MILP). It is currently based on the LP modeler PuLP.

It is an Open Source project located on GitLab at OMEGAlpes Gitlab An associated web interface is available at OMEGAlpes interface to help generate scripts

CHAPTER 2

Contents

2.1 About OMEGAlpes

OMEGAlpes is a linear optimization tool designed to easily generate multi-carrier energy system models. Its purpose is to assist in developing district energy projects by integrating design and operation in pre-studies phases. OMEGAlpes is open-source and written in Python with the licence **'Apache 2.0'_**. It is a generation model tool based on an intuitive and extensible object-oriented library that aims to provide a panel of pre-built energy units with predefined operational options, and associated constraints and objectives taking into account stakeholders.

OMEGAlpes is developed in order to provide various features regarding energy system optimization, and especially energy flexibility:

2.1.1 Energy flexibility features

To summarise the flexibility capabilities of Energy Units in OMEGAlpes:

- You can directly use the following parameters when creating an Energy Unit:
- min_time_on / min_time_off: to define the minimum time the unit must be operating / not operating once started up / once switched off.
- availability_hours: to define the number of hours the energy unit is available on the whole time horizon.
- e_min: to set the minimal energy the unit must consume during the time horizon
- e_max: to set the maximal energy the unit can consume during the time horizon
- The ramp rates min_ramp_up, max_ramp_up, min_ramp_down, max_ramp_down
- As a user you can also define a Shiftable Energy unit, you need to choose a power profile that can be shifted in time (so shorter in time than your time horizon, for instance a 2 hours power profile of a washing machine cycle compared to a 24 hours time horizon) and to define if the operation of the energy unit is mandatory or not.
- Finally, you can use the functions:
- add_operating_time_range to define a range of hours during which the energy unit can be operated every day.

- add_energy_limits_on_time_period to define a minimal or maximal energy limit on a given time period of the studied horizon.

Some key articles in open access regarding OMEGAlpes and its use for flexibility studies:

- Residential energy flexibility in districts:
- An Approach to Study District Thermal Flexibility Using Generative Modeling from Existing Data, Pajot et al.
- Impact of Heat Pump Flexibility in a French Residential Eco-District, Pajot et al.
- Building Reduced Model for MILP Optimization, Application to Demand Response of Residential Buildings, Pajot et al.
- Industrial energy flexibility in districts:
- Data-driven Modeling of Building Consumption Profile for Optimal Flexibility, Application to Energy Intensive Industry, Pajot et al.

OMEGAlpes was also presented and used in several scientific publications that can be found here:

2.1.2 OMEGAlpes publications

OMEGAlpes was presented in details in the following publications:

- In the 2021 *energies* article OMEGAlpes, an Open-Source Optimisation Model Generation Tool to Support Energy Stakeholders at District Scale, where the tool and its associated use cases are presented in detail.
- In 2019 in the *Building Simulation* conference article An Optimization Modeler as an Efficient Tool for Design and Operation for City Energy Stakeholders and Decision Makers
- In French in 2019 in Camille Pajot PhD thesis OMEGAlpes Outil d'aide à la décision pour une planification énergétique multi-fluides optimale à l'échelle des quartiers

The tool was also used in various scientific works:

- Pajot, C.; Delinchant, B.; Marechal, Y.; Wurtz, F.; Morriet, L.; Vincent, B.; Debray, F. Industrial Optimal Operation Plan-ning with Financial and Ecological Objectives. In Proceedings of the 7th International Conference on Smart Cities and Green ICT Systems; Funchal, Portugal, 16–18 March 2018; SciTePress—Science and and Technology Publications, Setúbal, Portugal, 2018; pp. 214–222.
- Pajot, C.; Nguyen, Q.; Delinchant, B.; Maréchal, Y.; Wurtz, F.; Robin, S.; Vincent, B.; Debray, F. Data-driven Modeling of Building Consumption Profile for Optimal Flexibility: Application to Energy Intensive Industry. In Proceedings of the Building Simulation Conference, Rome, Italy, 2–4 September 2019. Available online: https://hal.archives-ouvertes.fr/hal-02364669 (accessed on 17 December 2019).
- Hodencq, S.; Debray, F.; Trophime, C.; Vincent, B.; Stutz, B.; Delinchant, B. Thermohydraulics of High Field Magnets: From microns to urban community scale. In Proceedings of the 24ème Congrès Français de Mécanique, Brest, France, 26–30 August 2019.
- Pajot, C.; Artiges, N.; Delinchant, B.; Rouchier, S.; Wurtz, F.; Maréchal, Y. An Approach to Study District Thermal Flexibility Using Generative Modeling from Existing Data. Energies 2019, 12, 3632, doi:10.3390/en12193632.
- Morriet, L.; Debizet, G.; Wurtz, F. Multi-Actor Modelling for MILP Energy Systems Optimisation: Application to Collective Self-Consumption. In Proceedings of the Building Simulation 2019: 16th Conference of IBPSA, Rome, Italy, 2–4 September 2019. https://hal.archives-ouvertes.fr/hal-02285965
- Fitó, J.; Ramousse, J.; Hodencq, S.; Wurtz, F. Energy, exergy, economic and exergoeconomic (4E) multicriteria analysis of an industrial waste heat valorization system through district heating. Sustain. Energy Technol. Assess. 2020, 42, 100894, doi:10.1016/j.seta.2020.100894

- Fitó, J.; Hodencq, S.; Ramousse, J.; Wurtz, F.; Stutz, B.; Debray, F.; Vincent, B. Energy- and exergy-based optimal designs of a low-temperature industrial waste heat recovery system in district heating. Energy Convers. Manag. 2020, 211, 112753, doi:10.1016/j.enconman.2020.112753.
- Fitó, J.; Ramousse, J.; Hodencq, S.; Morriet, L.; Wurtz, F.; Debizet, G. Analyse technico-économique multi-acteurs de la conception d'un système de valorisation de chaleur fatale sur réseau de chaleur. In Proceedings of the Communautés Énergétiques, Autoproduction, Autoconsommation: Cadrages, Pratiques et Outils, Paris, France, 16 June 2020; p. 13.
- Hodencq, S.; Morriet, L.; Wurtz, F.; Delinchant, B.; Vincent, B.; Debray, F. Science ouverte pour l'optimisation de systèmes énergétiques: Des données et modèles ouverts à une infrastructure de recherche ouverte. In Proceedings of the Conférence IBPSA, Reims, France, 13–14 May 2020. Available online: https://hal.archives-ouvertes.fr/hal-03290009 (ac-cessed on 23 July 2021).
- Hodencq, S.; Fitó, J.; Debray, F.; Vincent, B.; Ramousse, J.; Delinchant, B.; Wurtz, F. Flexible waste heat management and recovery for an electro-intensive industrial process through energy/exergy criteria. In Proceedings of the Ecos 2021-The 34rth International Conference on Efficiency, Cost, Optimization, Simulation and Environmental Impact of Energy Systems, Taormina, Italy, 28 Jun–2 July 2021. Available online: https://hal.archives-ouvertes.fr/hal-03290126 (accessed on 23 July 2021).
- Hodencq, S.; Delinchant, B.; Frederic, W.; Artiges, N.; Ferrari, J.; Laranjeira, T.; Morriet, L.; Benjamin, V.; Francois, D. Towards an energy open science approach at district level: Application to Grenoble Presqu'île'. In Proceedings of the 1st International Workshop on Open Design & Open Source Hardware Product Development, Grenoble, France, 5–6 March 2020. Available online: https://hal.archives-ouvertes.fr/hal-03052326 (accessed on 1 March 2021).
- Hodencq, S.; Delinchant, B.; Wurtz, F., « Open and Reproducible Use Cases for Energy (ORUCE) methodology in systems design and operation: a dwelling photovoltaic self-consumption example », In Proceedings of the Building Simulation 2021: 17th Conference of IBPSA, Bruges, Belgium, 1 - 3 sept. 2021. https://hal.archives-ouvertes.fr/hal-03341883

2.1.3 Partners:

OMEGAlpes is mainly developped in the Grenoble Elctrical Engineering Laboratory (University Grenoble Alpes, CNRS, Grenoble INP, G2Elab, F-38000 Grenoble, France).

Other teams took part in the tool development:

- LOCIE (Laboratoire Optimisation de la Conception et Ingénierie de l'Environnement (LOCIE), CNRS UMR 5271—Université Savoie Mont Blanc, Polytech Annecy-Chambéry, Campus Scientifique, Savoie Technolac, CEDEX, 73376 Le Bourget-Du-Lac,) in thermal engineering for the exergy package in particular
- PACTE (Université Grenoble-Alpes, UMR 5194 PACTE) in social sciences for the actors package in particular
- MIAGE (Master Méthodes informatiques appliquées à la gestion des entreprises, Université Grenoble-Alpes) master students in computing for the GUI development.

2.2 OMEGAlpes Installation

2.2.1 Install OMEGAlpes

Do not hesitate to listen to a really nice music to be sure... ... it's going to work!

Python 3.6.0

Please use Python 3.6.0 for the project interpreter: Python 3.6

pip install omegalpes

Please install OMEGAlpes Lib with pip using on of the following the command prompt:

- **If you are admin on Windows or working on a virtual environment**:

```
pip install omegalpes
```

- **If you want a local installation or you are not admin**:

```
pip install --user omegalpes
```

- **If you are admin on Linux**:

```
sudo pip install omegalpes
```

Then, you can download (or clone) the OMEGAlpes Examples folder (repository) at : OMEGAlpes Examples Make shure that the name of the examples folder is: "omegalpes_examples".

Launch the examples (with Pycharm for instance) to understand how the OMEGAlpes Lib works. Remember that the examples are presented at : OMEGAlpes Examples Documentation

Enjoy your time using OMEGAlpes !

2.2.2 Other installation requirements

If the music was enough catchy, the following libraries should be already installed. If not, increase the volume and install the following libraries with the help below.

- **PuLP >= 2.1**

PuLP is an LP modeler written in python. PuLP can generate MPS or LP files and call GLPK, COIN CLP/CBC, CPLEX, and GUROBI to solve linear problems : PuLP

- **Matplotlib >= 2.2.2**

Matplotlib is a Python 2D plotting library : Matplotlib

- **Numpy >= 1.14.2**

NumPy is the fundamental package needed for scientific computing with Python. Numpy

- **Pandas >= 0.22.0**

Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. Pandas

— Command lover —

```
pip install <library_name>==version
```

If required, the command to upgrade the library is

```
pip install --upgrade <library_name>
```

— Pycharm lover —

Install automatically the library using pip with Pycharm on “File”, “settings…”, “Project Interpreter”, “+”, and choosing the required library

2.2.3 Install OMEGAlpes as a developer

Installation as a developer and local branch creation

Absolute silence, keep calm and stay focus… you can do it! :https://www.youtube.com/watch?v=g4mHPeMGTJM

1. Create a new folder in the suitable path, name it as you wish for instance : OMEGAlpes
2. Clone the OMEGAlpes library repository

   — Command lover —

   ```
   git clone https://gricad-gitlab.univ-grenoble-alpes.fr/omegalpes/
   ↪omegalpes.git
   ```

   — Pycharm lover —

   Open Pycharm
   On the Pycharm window, click on “Check out from version control” then choose “Git”.
   A “clone repository” window open.
   Copy the following link into the URL corresponding area:

   https://gricad-gitlab.univ-grenoble-alpes.fr/omegalpes/omegalpes.git

   Copy the path of the new folder created just before.
   Test if the connection to the git works and if it works click on “Clone”.
   Once OMEGAlpes is cloned, you must be able to see the full OMEGAlpes library on Pycharm or on another development environment.

   If the connection does not work and if you are working with local protected network, please try again with the wifi.
3. First, choose or change your project interpreter

   — Pycharm lover —

   Click on the yellow warning link or go to “File”, “settings…”, “Project Interpreter”

   You can:

   - either select the “Python 3.6” project interpreter but you may change the version of some library that you could use for another application.
   - either create a virtual environment in order to avoid this problem (recommended).

     Click on the star wheel near the project interpreter box.

Click on “add…”.
Select “New environment” if it not selected.
The location is pre-filled, if not fill it with the path of the folder as folder_path/venv
Select “Python 3.6” as your base interpreter
Then click on “Ok”

4. You can install the library on developing mode using the following command in command prompt once your are located it on the former folder. If you are calling OMEGAlpes library in another project, the following command enables you to refer to the OMEGAlpes library you are developing:

```
python setup.py develop
```

5. If it is not already done, install the library requirements.

   — Command lover —

   ```
   pip install <library_name>
   ```

   If required, the command to upgrade the library is

   ```
   pip install --upgrade <library_name>
   ```

   — Pycharm lover —

   You should still have a yellow warning. You can:

   - install automatically the libraries clicking on the yellow bar.
   - install automatically the library using pip with Pycharm on “File”, “settings…”, “Project Interpreter”, “+”, and choose the required library as indicated in the Library Installation Requirements part.

6. Finally, you can create your own local development branch.

   — Command lover —

   ```
   git branch <branch_name>
   ```

   — Pycharm lover —

   By default you are on a local branch named master.
   Click on “Git: master” located on the bottom write of Pycharm
   Select “+ New Branch”
   Name the branch as you convenience for instance “dev_your_name”

7. Do not forget to “rebase” regularly to update your version of the library.

   — Command lover —

   ```
   git rebase origin
   ```

   — Pycharm lover —

   To do so, click on your branch name on the bottom write of the Pycharm window select “Origin/master” and click on “Rebase current onto selected”

If you want to have access to examples and study cases, download (or clone) the OMEGAlpes Examples folder (repository) from : OMEGAlpes Examples . Make shure that the name of the examples folder is: “omegalpes_examples”. Remember that the examples are presented at : OMEGAlpes Examples Documentation

Enjoy your time developing OMEGAlpes!

2.3 OMEGAlpes Structure

The models are bases on **general**, **energy** and **actor** classes. The structure of the library is described on the following class diagrams:

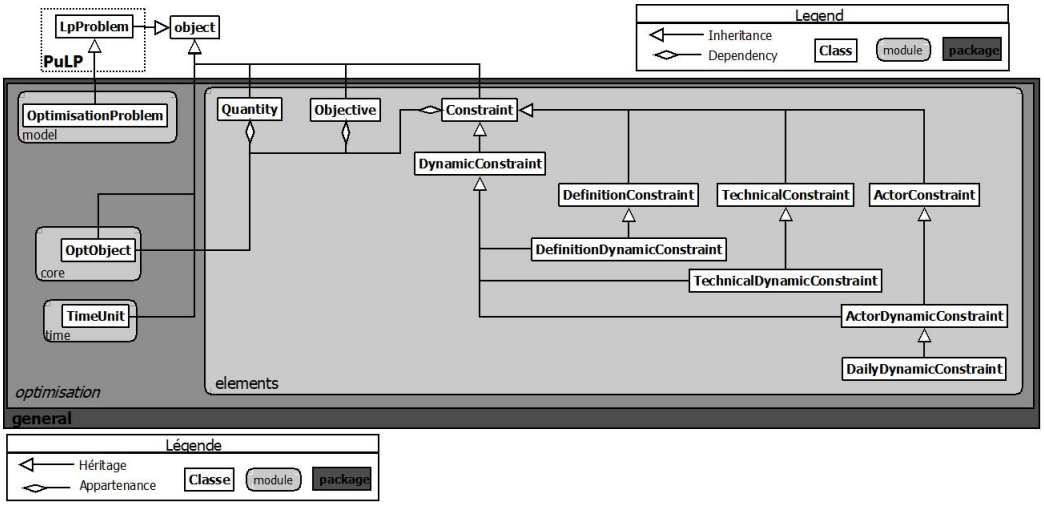

Fig. 1: *Figure: OMEGAlpes general class diagram*

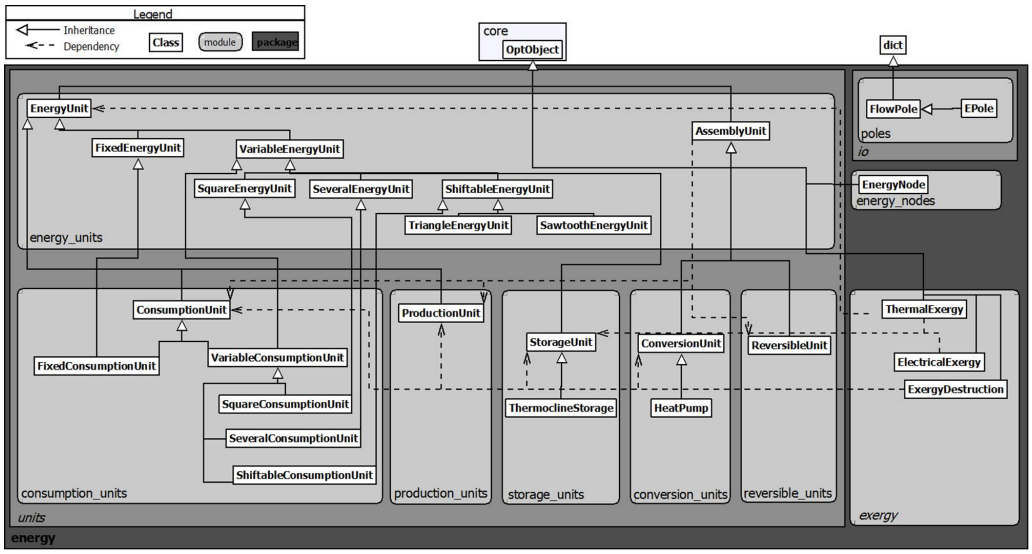

Fig. 2: *Figure: OMEGAlpes energy class diagram*

The energy units are detailed in the energy package

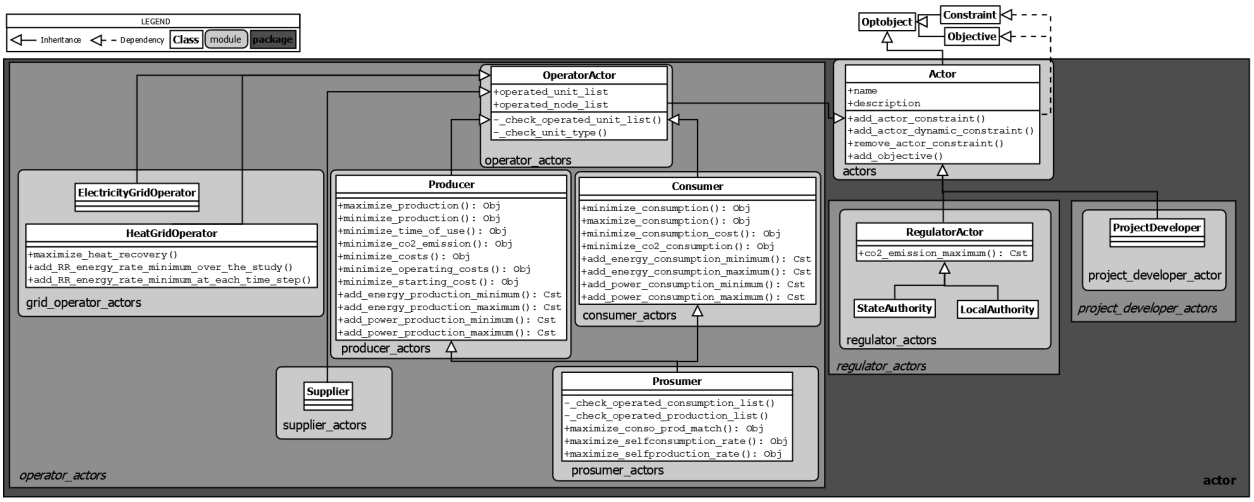

Fig. 3: *Figure: OMEGAlpes actor class diagram*

2.3.1 energy package

The energy package gathers the energy units, poles and nodes modules:

- *Energy_units module*
- *Consumption_units module*
- *Production_units module*
- *Conversion_units module*
- *Storage_units module*
- *Reversible_units module*
- *Energy_nodes module*
- *Thermal Building module*
- *Exergy module*
- *Poles module*

The energy units inherit from the `EnergyUnit` object which itself inherit from the `Unit` object.

Energy_units module

This module defines the energy units of OMEGAlpes. The production, consumption and storage unit will inherit from it.

The energy_units module defines the basic attributes and methods of an energy unit in OMEGAlpes.

The class EnergyUnit includes the following attributes and quantities:

- time: instance of TimeUnit describing the studied time period.
- energy_type: energy type of the energy unit (see energy.energy_types)
- p: instantaneous power of the energy unit (kW)

- e_tot: total energy either consumed or produced by the energy unit on

the studied time period during the time period (kWh) - u: binary describing if the unit is operating (1) or not (0) at t (i.e. delivering or consuming power) - poles: energy poles of the energy unit (see energy.io.poles)

EnergyUnit parameters can be used to add energy constraints or new attributes calculation to the energy unit, such as e_max or starting_cost.

This module also includes the classes: - FixedEnergyUnit: energy unit with a fixed power profile

- VariableEnergyUnit: energy unit with a variable power profile
- SquareEnergyUnit: energy unit with a defined square power profile,

inheriting from VariableEnergyUnit.

- ShiftableEnergyUnit: energy unit with a power profile that can be time

shifted, inheriting from VariableEnergyUnit.

- TriangleEnergyUnit: energy unit with a defined triangular power profile,

inheriting from VariableEnergyUnit.

- SawtoothEnergyUnit: energy unit with a defined sawtooth power profile,

inheriting from VariableEnergyUnit.

- SeveralEnergyUnit: Energy unit based on a fixed power curve enabling to

multiply several times (nb_unit) the same power curve.

- AssemblyUnit: an assembly unit has at least a production unit and a

consumption unit and is using one or several energy types. It can also integrate reversible energy units. It inherits from OptObject and it is the parent class of ConversionUnit and ReversibleUnit.

class omegalpes.energy.units.energy_units.**AssemblyUnit**(*time*, *name*, *prod_units=None*, *cons_units=None*, *rev_units=None*, *verbose=True*)

Bases: *omegalpes.general.optimisation.core.OptObject*

Description

Simple Assembly unit: assembly units has at least a production unit and a consumption unit and is using one or several energy types. It can also integrate reversible energy units. It inherits from OptObject and it is the parent class of ConversionUnit and ReversibleUnit.

Attributes

- time: TimeUnit describing the studied time period
- prod_units: list of the production units in the assembly unit.
- cons_units: list of the consumption units in the assembly unit.
- rev_units: list of the reversible units in the assembly unit.
- poles: dictionary of the poles of the assembly unit

minimize_exergy_destruction(*weight=1*, *pareto=False*)

This is the main objective of any exergetic optimization.

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

class omegalpes.energy.units.energy_units.**EnergyUnit**(*time*, *name*, *flow_direction='in'*, *p=None*, *p_min=0*, *p_max=10000.0*, *e_min=None*, *e_max=None*, *starting_cost=None*, *operating_cost=None*, *min_time_on=None*, *min_time_off=None*, *max_ramp_up=None*, *max_ramp_down=None*, *co2_out=None*, *availability_hours=None*, *energy_type=None*, *no_warn=True*, *verbose=True*)

Bases: *omegalpes.general.optimisation.core.OptObject*

Description

Module dedicated to the parent class (EnergyUnit) of :

- production units
- consumption units
- storage units

add_energy_limits_on_time_period(*e_min=0*, *e_max=None*, *start='YYYY-MM-DD HH:MM:SS'*, *end='YYYY-MM-DD HH:MM:SS'*, *period_index=None*)

Add an energy limit during a defined time period

Parameters

- **e_min** – Minimal energy set during the time period (int or float)
- **e_max** – Maximal energy set during the time period (int or float)
- **start** – Date of start of the time period YYYY-MM-DD HH:MM:SS (str)
- **end** – Date of end of the time period YYYY-MM-DD HH:MM:SS (str)

add_operating_time_range(*operating_time_range: [[<class 'str'>, <class 'str'>]]*)

Add a range of hours during which the energy unit can be operated. The final time should be greater than the initial time within a time range, except when the final time is '00:00'.

example: add_operating_time_range([['10:00', '12:00'], ['14:00', '17:00']])

Parameters **operating_time_range** – list of lists of strings in the format

HH:MM [[first hour operating: str, hour to stop (not operating): str], [second hour operating: str, hour to stop (not operating): str], etc]

NB: the previous version of add_operating_time_range (deprecated since version 0.3.1) had integers instead of str hours as parameters, do not forget to update it if needed !

minimize_co2_emissions(*weight=1*, *pareto=False*)

Objective to minimize the co2 emissions of the energy unit

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_costs(*weight=1*, *pareto=False*)
Objective to minimize the costs (starting and operating costs)

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_energy(*weight=1*, *pareto=False*)
Objective to minimize the energy of the energy unit

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_exergy(*energy_unit=None*, *weight=1*, *pareto=False*)
Alternate objective of exergy optimization that may be interesting in some cases.

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_exergy_destruction(*weight=1*, *pareto=False*)
This is the main objective of any exergetic optimization.

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_operating_cost(*weight=1*, *pareto=False*)
Objective to minimize the operating costs

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_starting_cost(*weight=1*, *pareto=False*)
Objective to minimize the starting costs

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_time_of_use(*weight=1*, *pareto=False*)
Objective to minimize the time of running of the energy unit

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

set_operating_time_range(*operating_time_range: [[<class 'str'>, <class 'str'>]]*)
DEPRECATED: the name of the function changed to add_operating_time_range for code consistency, please use this function !

Add a range of hours during which the energy unit can be operated. The final time should be greater than the initial time within a time range, except when the final time is '00:00'.

example: set_operating_time_range([['10:00', '12:00'], ['14:00', '17:00']])

Parameters **operating_time_range** – list of lists of strings in the format

HH:MM [[first hour operating: str, hour to stop (not operating): str], [second hour operating: str, hour to stop (not operating): str], etc]

class omegalpes.energy.units.energy_units.**FixedEnergyUnit**(*time*, *name: str*, *p: list*, *flow_direction='in'*, *starting_cost=None*, *operating_cost=None*, *co2_out=None*, *energy_type=None*, *verbose=True*)
Bases: *omegalpes.energy.units.energy_units.EnergyUnit*

Description

Energy unit with a fixed power profile.

Attributes

- p : instantaneous power known by advance (kW)
- energy_type : type of energy ('Electrical', 'Heat', ...)

class omegalpes.energy.units.energy_units.**SawtoothEnergyUnit**(*time*, *name*, *flow_direction*, *p_peak*, *p_low*, *alpha_peak*, *t_triangle*, *t_sawtooth*, *mandatory=True*, *starting_cost=None*, *operating_cost=None*, *co2_out=None*, *energy_type=None*, *verbose=True*)
Bases: *omegalpes.energy.units.energy_units.ShiftableEnergyUnit*

class omegalpes.energy.units.energy_units.**SeveralEnergyUnit**(*time*, *name*, *fixed_power*, *p_min=0*, *p_max=100000.0*, *imaginary=False*, *e_min=None*, *e_max=None*, *nb_unit_min=0*, *nb_unit_max=None*, *flow_direction='in'*, *starting_cost=None*, *operating_cost=None*, *max_ramp_up=None*, *max_ramp_down=None*, *co2_out=None*, *energy_type=None*, *verbose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.energy_units.VariableEnergyUnit*

Description

Energy unit based on a fixed power curve enabling to multiply several times (nb_unit) the same power curve.

Be careful, if imaginary == True, the solution may be imaginary as nb_unit can be continuous. The accurate number of the power unit should be calculated later

Attributes

- fixed_power : fixed power curve

class omegalpes.energy.units.energy_units.**ShiftableEnergyUnit**(*time*, *name: str*, *flow_direction*, *power_values: list*, *mandatory=True*, *co2_out=None*, *starting_cost=None*, *operating_cost=None*, *energy_type=None*, *verbose=True*)

Bases: *omegalpes.energy.units.energy_units.VariableEnergyUnit*

Description

EnergyUnit with shiftable power profile.

Attributes

- power_values : power profile to shift (kW)
- mandatory : indicates if the power is mandatory (True) or not (False)
- starting_cost : cost of the starting of the EnergyUnit
- operating_cost : cost of the operation (€/kW)

- energy_type : type of energy ('Electrical', 'Heat', ...)

class omegalpes.energy.units.energy_units.**SquareEnergyUnit**(*time*, *name*, *p_square*, *n_square*, *t_between_sq*, *t_square=1*, *flow_direction='in'*, *starting_cost=None*, *operating_cost=None*, *co2_out=None*, *energy_type=None*, *verbose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.energy_units.VariableEnergyUnit*

class omegalpes.energy.units.energy_units.**TriangleEnergyUnit**(*time*, *name*, *flow_direction*, *p_peak*, *alpha_peak*, *t_triangle: list*, *mandatory=True*, *starting_cost=None*, *operating_cost=None*, *co2_out=None*, *energy_type=None*, *verbose=True*)

Bases: *omegalpes.energy.units.energy_units.ShiftableEnergyUnit*

class omegalpes.energy.units.energy_units.**VariableEnergyUnit**(*time*, *name*, *flow_direction='in'*, *p_min=0*, *p_max=10000.0*, *e_min=None*, *e_max=None*, *starting_cost=None*, *operating_cost=None*, *min_time_on=None*, *min_time_off=None*, *max_ramp_up=None*, *max_ramp_down=None*, *co2_out=None*, *availability_hours=None*, *energy_type=None*, *verbose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.energy_units.EnergyUnit*

Consumption_units module

This module defines the consumption units

The consumption_units module defines various classes of consumption units, from generic to specific ones.

It includes :

- ConsumptionUnit : simple consumption unit. It inherits from EnergyUnit, its power flow direction is always 'in'.

  3 Objectives are also available :

  - minimize consumption,
  - maximize consumption,
  - minimize consumption costs.

- FixedConsumptionUnit : consumption with a fixed load profile. It inherits from ConsumptionUnit.
- VariableConsumptionUnit : consumption unit allowing for a variation of power between p_min et p_max. It inherits from ConsumptionUnit.

And also :

- SeveralConsumptionUnit: Consumption unit based on a fixed consumption curve enabling to multiply several times (nb_unit) the same consumption profile
- SeveralImaginaryConsumptionUnit: Consumption unit based on a fixed consumption curve enabling to multiply several times (nb_unit) the same consumption profile. Be careful, the solution may be imaginary as nb_unit can be continuous. The accurate number of the Consumption units should be calculated later
- SquareConsumptionUnit: Consumption unit with a fixed value and fixed duration.
- ShiftableConsumptionUnit: Consumption unit with shiftable consumption profile.

class omegalpes.energy.units.consumption_units.**ConsumptionUnit**(*time*, *name*, *p=None*, *p_min=0*, *p_max=100000.0*, *e_min=None*, *e_max=None*, *co2_out=None*, *starting_cost=None*, *consumption_cost=None*, *min_time_on=None*, *min_time_off=None*, *max_ramp_up=None*, *max_ramp_down=None*, *availability_hours=None*, *energy_type=None*, *verbose=True*)

Bases: *omegalpes.energy.units.energy_units.EnergyUnit*

Description

Simple Consumption unit. The parameters and attributes are described in EnergyUnit parent class. Here, consumption_cost is the cost associated to the energy consumption of the unit (€/kWh).

maximize_consumption(*weight=1*, *pareto=False*)

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_consumption(*weight=1*, *pareto=False*)

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_consumption_cost(*weight=1*, *pareto=False*)

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

class omegalpes.energy.units.consumption_units.**FixedConsumptionUnit**(*time*, *name*, *p: list = None*, *co2_out=None*, *starting_cost=None*, *operating_cost=None*, *energy_type=None*, *verbose=True*)

Bases: *omegalpes.energy.units.energy_units.FixedEnergyUnit*, *omegalpes.energy.units.consumption_units.ConsumptionUnit*

Description

Consumption unit with a fixed consumption profile.

Attributes

- p : instantaneous power demand known in advance (kW)
- energy_type : type of energy ('Electrical', 'Heat', ...)
- consumption_cost : cost associated to the energy consumption

class omegalpes.energy.units.consumption_units.**SeveralConsumptionUnit**(*time*, *name*, *fixed_cons*, *imag-i-nary=False*, *p_min=0*, *p_max=100000.0*, *e_min=None*, *e_max=None*, *nb_unit_min=0*, *nb_unit_max=None*, *co2_out=None*, *start-ing_cost=None*, *op-erat-ing_cost=None*, *max_ramp_up=None*, *max_ramp_down=None*, *en-ergy_type=None*, *ver-bose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.consumption_units.VariableConsumptionUnit*, *omegalpes.energy.units.energy_units.SeveralEnergyUnit*

Description

Consumption unit based on a fixed consumption curve enabling to multiply several times (nb_unit) the same consumption curve.

Attributes

- fixed_cons : fixed consumption curve

class omegalpes.energy.units.consumption_units.**ShiftableConsumptionUnit**(*time*, *name: str*, *power_values*, *manda-tory=True*, *co2_out=None*, *start-ing_cost=None*, *op-er-at-ing_cost=None*, *en-ergy_type=None*, *ver-bose=True*)

Bases: *omegalpes.energy.units.energy_units.ShiftableEnergyUnit*, *omegalpes.energy.units.consumption_units.VariableConsumptionUnit*

Description

Consumption unit with shiftable consumption profile.

Attributes

- power_values : consumption profile to shift (kW)
- mandatory : indicates if the consumption is mandatory (True) or not

(False) * starting_cost : cost of the starting of the consumption * operating_cost : cost of the operation (€/kW) * energy_type : type of energy ('Electrical', 'Heat', ...)

class omegalpes.energy.units.consumption_units.**SquareConsumptionUnit**(*time*, *name*, *p_square*, *duration*, *n_square*, *t_between_sq*, *co2_out=None*, *starting_cost=None*, *operating_cost=None*, *energy_type=None*, *verbose=True*, *no_warn=False*)

Bases: *omegalpes.energy.units.energy_units.SquareEnergyUnit*, *omegalpes.energy.units.consumption_units.VariableConsumptionUnit*

Description

Consumption unit with a fixed value and fixed duration.
Only the time of beginning can be modified
Operation can be mandatory or not

Attributes

- p : instantaneous power consumption (kW)
- duration : duration of the power delivery (hours)
- mandatory : indicates if the power delivery is mandatory or not
- energy_type : type of energy ('Electrical', 'Heat', ...)
- consumption_cost : cost associated to the energy consumption

class omegalpes.energy.units.consumption_units.**VariableConsumptionUnit**(*time*, *name*, *p_min=0*, *p_max=100000.0*, *e_min=None*, *e_max=None*, *co2_out=None*, *starting_cost=None*, *operating_cost=None*, *min_time_on=None*, *min_time_off=None*, *max_ramp_up=None*, *max_ramp_down=None*, *energy_type=None*, *verbose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.energy_units.VariableEnergyUnit*, *omegalpes.energy.units.consumption_units.ConsumptionUnit*

Description

Consumption unit with a variation of power between p_min et p_max.

Attributes

- p_max : maximal instantaneous power consumption (kW)
- p_min : minimal instantaneous power consumption (kW)
- energy_type : type of energy ('Electrical', 'Heat', ...)

Production_units module

This module defines the production units

The production_units module defines various kinds of production units with associated attributes and methods.

It includes :

- ProductionUnit : simple production unit inheriting from EnergyUnit and with an outer flow direction. The outside co2 emissions, the starting cost, the operating cost, the minimal operating time, the minimal non-operating time, the maximal increasing ramp and the maximal decreasing ramp can be filled.

  Objectives are also available :

  - minimize starting cost, operating cost, total cost
  - minimize production, co2_emissions, time of use
  - maximize production
- FixedProductionUnit : Production unit with a fixed production profile.
- VariableProductionUnit : Production unit with a variation of power between p_min et p_max.

And also :

- SeveralProductionUnit: Production unit based on a fixed production curve enabling to multiply several times (nb_unit) the same production curve
- SeveralImaginaryProductionUnit: Production unit based on a fixed production curve enabling to multiply several times (nb_unit) the same production curve. Be careful, the solution may be imaginary as nb_unit can be continuous. The accurate number of the production unit should be calculated later
- SquareProductionUnit: Production unit with a fixed value and fixed duration.
- ShiftableProductionUnit: Production unit with shiftable production profile.

class omegalpes.energy.units.production_units.**FixedProductionUnit**(*time*, *name: str*, *p: list = None*, *co2_out=None*, *particle_emission=None*, *starting_cost=None*, *operating_cost=None*, *energy_type=None*, *rr_energy=False*, *verbose=True*)

Bases: *omegalpes.energy.units.energy_units.FixedEnergyUnit*, *omegalpes.energy.units.production_units.ProductionUnit*

Description

Production unit with a fixed production profile.

Attributes

- p : instantaneous power production known by advance (kW)
- energy_type : type of energy ('Electrical', 'Heat', ...)

class omegalpes.energy.units.production_units.**ProductionUnit**(*time*, *name*, *p=None*, *p_min=0*, *p_max=100000.0*, *e_min=None*, *e_max=None*, *co2_out=None*, *particle_emission=None*, *starting_cost=None*, *operating_cost=None*, *min_time_on=None*, *min_time_off=None*, *max_ramp_up=None*, *max_ramp_down=None*, *availability_hours=None*, *energy_type=None*, *rr_energy=False*, *verbose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.energy_units.EnergyUnit*

Description

Simple Production unit. The parameters and attributes are described in EnergyUnit parent class.

maximize_production(*weight=1*, *pareto=False*)

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_production(*weight=1*, *pareto=False*)

Parameters

- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

class omegalpes.energy.units.production_units.**SeveralProductionUnit**(*time*, *name*, *fixed_prod*, *imaginary=False*, *p_min=0*, *p_max=100000.0*, *e_min=None*, *e_max=None*, *nb_unit_min=0*, *nb_unit_max=None*, *co2_out=None*, *particle_emission=None*, *starting_cost=None*, *operating_cost=None*, *max_ramp_up=None*, *max_ramp_down=None*, *energy_type=None*, *rr_energy=False*, *verbose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.production_units.VariableProductionUnit*, *omegalpes.energy.units.energy_units.SeveralEnergyUnit*

Description

Production unit based on a fixed production curve enabling to multiply several times (nb_unit) the same production curve. nb_unit is an integer variable.

Attributes

- fixed_prod : fixed production curve

class omegalpes.energy.units.production_units.**ShiftableProductionUnit**(*time*, *name: str*, *power_values*, *mandatory=True*, *co2_out=None*, *particle_emission=None*, *starting_cost=None*, *operating_cost=None*, *energy_type=None*, *rr_energy=False*, *verbose=True*)

Bases: *omegalpes.energy.units.energy_units.ShiftableEnergyUnit*, *omegalpes.energy.units.production_units.VariableProductionUnit*

Description

Production unit with shiftable production profile.

Attributes

- power_values : production profile to shift (kW)
- mandatory : indicates if the production is mandatory : True or not : False
- starting_cost : cost of the starting of the production
- operating_cost : cost of the operation (€/kW)
- energy_type : type of energy ('Electrical', 'Heat', ...)

class omegalpes.energy.units.production_units.**SquareProductionUnit**(*time*, *name*, *p_square*, *duration*, *n_square*, *t_between_sq*, *co2_out=None*, *particle_emission=None*, *starting_cost=None*, *operating_cost=None*, *energy_type=None*, *rr_energy=False*, *verbose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.energy_units.SquareEnergyUnit*, *omegalpes.energy.units.production_units.VariableProductionUnit*

Description

Production unit with a fixed value and fixed duration.
Only the time of beginning can be modified.
Operation can be mandatory or not.

Attributes

- p : instantaneous power production (kW)
- duration : duration of the power delivery (hours)
- mandatory : indicates if the power delivery is mandatory or not
- energy_type : type of energy ('Electrical', 'Heat', ...)

class omegalpes.energy.units.production_units.**VariableProductionUnit**(*time*, *name*, *p_min=0*, *p_max=100000.0*, *e_min=None*, *e_max=None*, *co2_out=None*, *particle_emission=None*, *starting_cost=None*, *operating_cost=None*, *min_time_on=None*, *min_time_off=None*, *max_ramp_up=None*, *max_ramp_down=None*, *energy_type=None*, *rr_energy=False*, *verbose=True*, *no_warn=True*)

Bases: *omegalpes.energy.units.energy_units.VariableEnergyUnit*, *omegalpes.energy.units.production_units.ProductionUnit*

Description

Production unit with a variation of power between p_min et p_max.

Attributes

- p_max : maximal instantaneous power production (kW)
- p_min : minimal instantaneous power production (kW)
- energy_type : type of energy ('Electrical', 'Heat', ...)

Conversion_units module

This module defines the conversion units, inheriting from AssemblyUnits

The conversion_units module defines various classes of conversion units, from generic to specific ones.

It includes :

- ConversionUnit: simple conversion unit. It inherits from AssemblyUnit.
- SingleConversionUnit: Conversion unit made of a single consumption unit and a single production unit. It can be used for any energy vector, and an efficiency ratio between the input (consumption) and the output (production)
- ElectricalToThermalConversionUnit : Electrical to thermal Conversion unit with an electricity consumption and a thermal production linked by and electrical to thermal ratio. It inherits from ConversionUnit
- HeatPump : Simple Heat Pump with an electricity consumption, a heat production and a heat consumption. It has a theoretical coefficient of performance COP and inherits from ConversionUnit.

class omegalpes.energy.units.conversion_units.**ConversionUnit**(*time*, *name*, *prod_units=None*, *cons_units=None*, *rev_units=None*, *verbose=True*)

Bases: *omegalpes.energy.units.energy_units.AssemblyUnit*

Description

Simple Conversion unit, inheriting from AssemblyUnit

Attributes

- time : TimeUnit describing the studied time period
- prod_units : list of the production units
- cons_units : list of the consumption units
- poles : dictionary of the poles of the conversion unit

class omegalpes.energy.units.conversion_units.**ElectricalConversionUnit**(*time*, *name*, *pmin_in_elec=0*, *pmax_in_elec=100000.0*, *p_in_elec=None*, *pmin_out_elec=0*, *pmax_out_elec=100000.0*, *p_out_elec=None*, *elec_to_elec_ratio=1*)

Bases: *omegalpes.energy.units.conversion_units.ConversionUnit*

Description

Electrical Conversion unit with an electricity consumption and an electricity production

Attributes

- elec_production_unit : electricity production unit (electrical output)
- elec_consumption_unit : electricity consumption unit (electrical input)
- conversion : Definition Dynamic Constraint linking the electrical

input to the electrical output through the elec_to_elec ratio

class omegalpes.energy.units.conversion_units.**ElectricalToThermalConversionUnit**(*time*, *name*, *pmin_in_elec=*
pmax_in_elec=
p_in_elec=Non
pmin_out_therm
pmax_out_ther
p_out_therm=N
*elec_to_therm_*
verbose=True)

Bases: *omegalpes.energy.units.conversion_units.ConversionUnit*

Description DEPRECATED: please now use SingleConversionUnit with relevant energy types

Electrical to thermal Conversion unit with an electricity consumption and a thermal production

Attributes

- thermal_production_unit : thermal production unit (thermal output)
- elec_consumption_unit : electricity consumption unit (electrical input)
- conversion : Definition Dynamic Constraint linking the electrical

input to the thermal output through the electrical to thermal ratio

class omegalpes.energy.units.conversion_units.**HeatPump**(*time*, *name*, *pmin_in_elec=0*, *pmax_in_elec=100000.0*, *p_in_elec=None*, *pmin_in_therm=0*, *pmax_in_therm=100000.0*, *p_in_therm=None*, *pmin_out_therm=0*, *pmax_out_therm=100000.0*, *p_out_therm=None*, *cop=3*, *losses=0*, *min_time_on=None*)

Bases: *omegalpes.energy.units.conversion_units.ConversionUnit*

Description

Simple Heat Pump with an electricity consumption, a thermal production and a thermal consumption. It has a theoretical coefficient of performance COP and inherits from ConversionUnit.

Attributes

- thermal_production_unit : thermal production unit (condenser)
- elec_consumption_unit : electricity consumption unit (electrical input)
- thermal_consumption_unit : heay consumption unit (evaporator)
- COP : Quantity describing the coefficient of performance of the heat pump
- conversion : Definition Dynamic Constraint linking the electrical

input to the thermal output through the electrical to thermal ratio * power_flow : Definition Dynamic constraint linking the thermal output to the electrical and thermal inputs in relation to the losses.

class omegalpes.energy.units.conversion_units.**ReversibleConversionUnit**(*time*, *name*, *pmin_up=0*, *pmax_up=100000.0*, *pmin_down=0*, *pmax_down=100000.0*, *up2down_eff=1*, *down2up_eff=1*, *energy_type_up=None*, *energy_type_down=None*, *verbose=True*)

Bases: *omegalpes.energy.units.conversion_units.ConversionUnit*

Description

Reversible Conversion unit with two reversible units, one for each side of the conversion unit. Theses sides will be called upstream and downstream.

Attributes

- rev_unit_upstream: reversible unit upstream
- rev_unit_downstream: reversible unit downstream
- conversion_up2down: Definition Dynamic Constraint linking the

consumption of the upstream reversible unit to the production of the downstream reversible unit through the up2down_eff * conversion_down2up: Definition Dynamic Constraint linking the consumption of the downstream reversible unit to the production of the upstream reversible unit through the down2up_eff

class omegalpes.energy.units.conversion_units.**SingleConversionUnit**(*time*, *name*, *pmin_in=0*, *pmax_in=100000.0*, *p_in=None*, *energy_type_in=None*, *pmin_out=0*, *pmax_out=100000.0*, *p_out=None*, *energy_type_out=None*, *efficiency_ratio=1*, *verbose=True*)

Bases: *omegalpes.energy.units.conversion_units.ConversionUnit*

Description

Conversion unit made of a single consumption unit and a single production unit. It can be used for any energy vector, and an efficiency ratio between the input (consumption) and the output (production)

Attributes * production_unit: output of the conversion unit * consumption_unit: input of the conversion unit * conversion: Definition Dynamic Constraint linking the input to the output through a ratio

Storage_units module

This module defines the storage units

The storage_units module defines various kinds of storage units with associated attributes and methods, from simple to specific ones.

It includes :

- StorageUnit: simple storage unit inheriting from EnergyUnit, with storage specific attributes. It includes the objective "minimize capacity".
- Thermocline storage: a thermal storage that need to cycle (i.e. reach SOC_max) every period of Tcycle

class omegalpes.energy.units.storage_units.**StorageUnit**(*time*, *name*, *pc_min=0*, *pc_max=None*, *pd_min=0*, *pd_max=None*, *capacity=None*, *e_0=None*, *e_f=None*, *soc_min=0*, *soc_max=1*, *eff_c=1*, *eff_d=1*, *self_disch=0*, *self_disch_t=0*, *ef_is_e0=False*, *cycles=None*, *energy_type=None*, *e_min_ch=None*, *e_max_ch=None*, *e_min_disch=None*, *e_max_disch=None*)

Bases: *omegalpes.energy.units.energy_units.AssemblyUnit*

Description

Simple Storage unit

If storage capacity isn't an optimization variable, please assign a capacity value.

Attributes

- charge (VariableConsumptionUnit) : represents the charge part of the

storage unit * discharge (VariableProductionUnit) : represents the discharge part of the storage unit * capacity (Quantity): maximal energy that can be stored [kWh] * e (Quantity): energy at time t in the storage [kWh] * p (Quantity): power at time t in the storage [kW] * e (Quantity): energy at time t in the storage [Binary] * set_soc_min (TechnicalDynamicConstraint): constraining the energy to be

above the value : soc_min*capacity

- set_soc_max (TechnicalDynamicConstraint): constraining the energy to be below the value : soc_max*capacity storage unit : 0 : Not charging & 1 : charging
- calc_e (DefinitionDynamicConstraint) : energy calculation at time t ; relation power/energy
- calc_p (DefinitionDynamicConstraint) : power calculation at time t ;

power flow equals charging power minus discharging power

- on_off_stor (DefinitionDynamicConstraint) : making u[t] matching with storage modes (on/off)
- set_e_0 (ActorConstraint) : set the energy state for t=0
- e_f (Quantity) : energy in the storage at the end of the time horizon, i.e. after the last time step [kWh]
- e_f_min (TechnicalConstraint) : e_f value is constrained above soc_min*capacity
- e_f_max (TechnicalConstraint) : e_f value is constrained below soc_max*capacity
- set_e_f (ActorConstraint) : when e_f is given, it is set in the same way the energy is, but after the last time step
- calc_e_f (DefinitionConstraint) : when e_f is not given, it is calculated in the same way the energy is, but after the last time step
- ef_is_e0 (TechnicalConstraint) : Imposing ef=e0 on the time period.
- cycles (TechnicalDynamicConstraint) : setting a cycle constraint e[t] = e[t+cycles/dt]

minimize_capacity(*pc_max_ratio: float = None*, *pd_max_ratio: float = None*, *weight=1*)
Objective of minimizing the capacity. If pc_max_ratio and pd_max_ratio are set AND pc_max and pd_max have None value in the StorageUnit, pc_max and pd_max are constrained to have values in accordance with the given ratio of the capacity.

Parameters **weight** – Weight coefficient for the objective

:param pc_max_ratio : ratio of the capacity for pc_max value i.e. if pc_max_ratio is 1/2, pc_max = capacity / 2. This ratio should be taken in accordance with the value of the time step. :param pd_max_ratio : ratio of the capacity for pd_max value i.e. if pd_max_ratio is 1/2, pd_max = capacity / 2. This ratio should be taken in accordance with the value of the time step.

class omegalpes.energy.units.storage_units.**StorageUnitTm1**(*time*, *name='StUtm1'*, *pc_min=0*, *pc_max=100000.0*, *pd_min=0*, *pd_max=100000.0*, *capacity=None*, *e_0=None*, *e_f=None*, *soc_min=0*, *soc_max=1*, *eff_c=1*, *eff_d=1*, *self_disch=0*, *self_disch_t=0*, *ef_is_e0=False*, *cycles=None*, *energy_type=None*, *operator=None*)

Bases: *omegalpes.energy.units.storage_units.StorageUnit*

Description Storage unit where the energy is described at the end of a timestep. Calculation : e[t]-e[t-1] = dt * (pc[t]*eff_c - pd[t]*1/eff_d - self_disch*capa - self_disch_t*e[t]) In this case, e_f is not defined as a quantity

Attributes

- capacity (Quantity): maximal energy that can be stored [kWh]
- e (Quantity): energy at time t in the storage [kWh]
- set_soc_min (DynamicConstraint): constraining the energy to be

above the value : soc_min*capacity * set_soc_max (DynamicConstraint): constraining the energy to be below the value : soc_max*capacity * pc (Quantity) : charging power [kW] * pd (Quantity) : discharging power [kW] * uc (Quantity) : binary variable describing the charge of the storage unit : 0 : Not charging & 1 : charging * calc_e (DynamicConstraint) : energy calculation at time t ; relation power/energy * calc_p (DynamicConstraint) : power calculation at time t ; power flow equals charging power minus discharging power * on_off_stor (DynamicConstraint) : making u[t] matching with storage modes (on/off) * def_max_charging (DynamicConstraint) : defining the max charging power, avoiding charging and discharging at the same time * def_max_discharging (DynamicConstraint) : defining the max discharging power, avoiding charging and discharging at the same time * def_min_charging (DynamicConstraint) : defining the min charging power, avoiding charging and discharging at the same time * def_min_discharging (DynamicConstraint) : defining the min discharging power, avoiding charging and discharging at the same time * set_e_0 (ExternalConstraint) : set the energy state for t=0 * set_e_f (ExternalConstraint) : set the energy state for the last time step * ef_is_e0 (ExternalConstraint) : Imposing ef=e0 on the time period. * cycles (ExternalDynamicConstraint) : setting a cycle constraint e[t] = e[t+cycles/dt]

class omegalpes.energy.units.storage_units.**ThermoclineStorage**(*time*, *name*, *pc_min=0*, *pc_max=100000.0*, *pd_min=0*, *pd_max=100000.0*, *capacity=None*, *e_0=None*, *e_f=None*, *soc_min=0*, *soc_max=1*, *eff_c=1*, *eff_d=1*, *self_disch=0*, *e_min_ch=None*, *e_max_ch=None*, *e_min_disch=None*, *e_max_disch=None*, *Tcycl=120*, *ef_is_e0=False*)

Bases: *omegalpes.energy.units.storage_units.StorageUnit*

Description

Class ThermoclineStorage : class defining a thermocline heat storage, inheriting from StorageUnit.

Attributes

- is_soc_max (Quantity) : indicating if the storage is fully charged 0:No 1:Yes
- def_is_soc_max_inf (DynamicConstraint) : setting the right value for is_soc_max
- def_is_soc_max_sup (DynamicConstraint) : setting the right value for is_soc_max
- force_soc_max (TechnicalDynamicConstraint) : The energy has to be

at least once at its maximal value during the period Tcycl.

Reversible_units module

This module defines the reversible units

The reversible_units module defines various kinds of reversible units with associated attributes and methods, from simple to specific ones, inheriting from AssemblyUnit.

It includes :

- ReversibleUnit : simple reversible unit with only one consumption and

one production units. It can both produce and consume energy but not at the same time.

class omegalpes.energy.units.reversible_units.**ReversibleUnit**(*time*, *name*, *pmin_cons=0*, *pmax_cons=100000.0*, *p_cons=None*, *pmin_prod=0*, *pmax_prod=100000.0*, *p_prod=None*, *energy_type_prod=None*, *energy_type_cons=None*, *verbose=True*)

Bases: *omegalpes.energy.units.energy_units.AssemblyUnit*

Description

Simple Reversible unit inheriting from AssemblyUnit. It is made of a consumption unit and a production unit that can both operate but not at the same time (reversible constraint).

Attributes

- production_unit (ProductionUnit)
- consumption_unit (ConsumptionUnit)
- def_rev (DefinitionDynamicConstraint): definition of the reversible

constraint * def_rev_c (DefinitionDynamicConstraint): definition of the reversible constraint in the case where only the consumption is fixed * def_rev_p (DefinitionDynamicConstraint): definition of the reversible constraint in the case where only the production is fixed

Energy_nodes module

This module defines the energy nodes that will allow energy transmission between the various energy units and conversion units

The energy_node module includes the EnergyNode class for energy transmission between production, consumption, conversion and storage. Defining several energy nodes and exporting/importing energy between them can also allow for a better demarcation of the energy system.

class omegalpes.energy.energy_nodes.**EnergyNode**(*time*, *name*, *energy_type=None*, *operator=None*)

Bases: *omegalpes.general.optimisation.core.OptObject*

This class defines an energy node.

add_connected_energy_unit(*unit*)
Add an EnergyUnit to the connected_units list

add_pole(*pole: omegalpes.energy.io.poles.Epole*) → None
Add an energy pole to the poles_list

Parameters **pole** – Epole

connect_units(**units*)
Connecting all EnergyUnit to the EnergyNode

Parameters **units** (EnergyUnit) – EnergyUnits connected to the EnergyNode

create_export(*node*, *export_min*, *export_max*)
Create the export from the EnergyNode (self) to the EnergyNode (node)

Parameters

- **node** – EnergyNode to whom power can be exported
- **export_min** – Minimal value of exported power when there is export
- **export_max** – Maximal value of exported power when there is export

Returns Quantity that defines the power exported

export_to_node(*node*, *export_min=0*, *export_max=100000.0*)
Add an export of power from the node to another node

Parameters

- **node** – EnergyNode to whom power can be exported
- **export_min** – Minimal value of exported power when there is export
- **export_max** – Maximal value of exported power when there is export

get_connected_energy_units
Return the list of connected EnergyUnits in the EnergyNode

get_exports
Return the list of exports to the EnergyNode

get_flows
Get all the power flows of the energy node :rtype: list :return: list of power flows

get_imports
Return the list of imports to the EnergyNode

get_input_poles

get_output_poles

get_poles
Return the list of energy poles in the EnergyNode

import_from_node(*node*, *import_min=0*, *import_max=100000.0*)

Parameters

- **node** – EnergyNode from whom power can be imported
- **import_min** – Minimal value of imported power when there is import
- **import_max** – Maximal value of imported power when there is import

is_export_flow(*flow*)
Get if the power flow is an export or not

is_import_flow(*flow*)
Get if the power flow is an import or not

set_power_balance()
Set the power balance equation for the EnergyNode

Thermal Building module

This module enables to model buildings as thermal loads

class omegalpes.energy.buildings.thermal.**HeatingLoad**(*time*, *name*, *tz*, *p_max=10000000000000.0*, *T_set=19*, *temp_margin=1*, *no_cooling=True*)

Bases: *omegalpes.energy.units.consumption_units.VariableConsumptionUnit*

maximize_thermal_comfort(*T_op=None*, *weight=1*)

Parameters

- **T_op** – Operative temperature wished for the maximal thermal comfort
- **weight** – Weight of the objective

class omegalpes.energy.buildings.thermal.**RCNetwork_1**(*time*, *name*, *T_ext*, *theta_ec*, *theta_em*, *T_int_min=0*, *T_int_max=50*, *theta_ea=None*, *theta_c=None*, *theta_m=None*, *h_ea=0*, *h_ac=0*, *h_ec=0*, *h_mc=0*, *h_em=0*, *c_m=0*, *f_im=None*, *f_r_l=0.7*, *f_r_p=0.5*, *f_r_a=0.2*, *f_sa=0.1*, *f_sm=None*, *f_sc=None*, *f_ic=None*, *f_hc_cv=None*, *U_wall=0.2*, *U_win=1.2*, *U_roof=0.2*, *e_wall=0.9*, *e_win=0.9*, *e_roof=0.9*, *a_wall=0.6*, *a_roof=0.6*, *A_wall=None*, *A_win=None*, *A_roof=None*, *owner=None*)

Bases: *omegalpes.general.optimisation.core.OptObject*

class omegalpes.energy.buildings.thermal.**ThermalZone**(*rc_network*, *phi_i_a*, *phi_i_l*, *phi_i_p*, *I_sol_av*, *Fsh_win*, *T_mean*, *T_dew*, *sky_cover=1*, *T_ext=None*, *hvac_prop=None*)

Bases: *omegalpes.general.optimisation.core.OptObject*

split_heating_and_cooling(*p_max_heating=10000000000000.0*, *p_max_cooling=10000000000000.0*)

class omegalpes.energy.buildings.thermal.**ZEA_RCNetwork_1**(*time*, *name*, *T_ext*, *A_f*, *A_win*, *Aext_v*, *A_roof*, *footprint*, *U_win*, *U_wall*, *U_roof*, *U_base*, *floors*, *e_wall=0.9*, *e_win=0.9*, *e_roof=0.9*, *a_wall=0.6*, *a_roof=0.6*, *construction='heavy'*, *height_bg=0*, *perimeter=0*, *f_hc_cv=1*, *void=0*, *hvac_prop=None*, *T_int_min=0*, *T_int_max=50*, *owner=None*)

Bases: *omegalpes.energy.buildings.thermal.RCNetwork_1*

omegalpes.energy.buildings.thermal.**calc_Am**(*Cm_Af*, *Af*)

omegalpes.energy.buildings.thermal.**calc_Aop_bel**(*height_bg*, *perimeter*, *footprint*)

omegalpes.energy.buildings.thermal.**calc_Aop_sup**(*Awall_all*, *void*, *window_to_wall_ratio*)

omegalpes.energy.buildings.thermal.**calc_Hd**(*Aop_sup*, *U_wall*, *footprint*, *U_roof*)

omegalpes.energy.buildings.thermal.**calc_Hg**(*Aop_bel*, *U_base*)

omegalpes.energy.buildings.thermal.**calc_Htr_op**(*Aop_bel*, *Aop_sup*, *footprint*, *U_base*, *U_wall*, *U_roof*)

omegalpes.energy.buildings.thermal.**calc_I_rad_linearization_coef**(*Tdry*, *Tdew*, *Tlin*, *sky_cover=1*)

Parameters

- **T_dry** – Dry bulb temperature in Celsius
- **T_dew** – Dew point temperature in Celsius
- **sky_cover** –

Returns list(A), list(B):

omegalpes.energy.buildings.thermal.**calc_I_sol**(*I_sol_average*, *Aop_sup*, *Aroof*, *Awin*, *a_wall*, *a_roof*, *U_wall*, *U_roof*, *Fsh_win*)

Parameters

- **I_sol_average** – W/m2
- **Aop_sup** – Opaque wall areas above ground (excluding voids and windows) [m2]
- **Aroof** – Roof area [m2]
- **Awin** – Windows area [m2]
- **a_wall** – Absorption coefficient of the walls [0..1]
- **a_roof** – Absorption coefficeint of the roof [0..1]
- **U_wall** –
- **U_roof** –
- **Fsh_win** – Shading factor for windows

Returns I_sol [kW]

omegalpes.energy.buildings.thermal.**calc_T_sky**(*T_dry*, *T_dew*, *sky_cover=1*)

Parameters

- **T_dry** – Dry bulb temperature in Celsius
- **T_dew** – Dew point temperature in Celsius
- **sky_cover** – Sky cover

omegalpes.energy.buildings.thermal.**calc_cm**(*Cm_Af*, *Af*)

omegalpes.energy.buildings.thermal.**calc_skytemp**(*Tdrybulb*, *Tdewpoint*, *N=1*)

Copyright 2014, Architecture and Building Systems - ETH Zurich

Parameters

- **Tdrybulb** – Drybuld temperature [°C]
- **Tdewpoint** – Dewpoint temperature [°C]
- **N** – Sky cover

Returns Sky temperature in °C

omegalpes.energy.buildings.thermal.**get_Cm_Af**(*construction*)

Description code Cm_Af Light construction T1 110000 Medium construction T2 165000 Heavy construction T3 300000

omegalpes.energy.buildings.thermal.**lookup_effective_mass_area_factor**(*cm*)

Look up the factor to multiply the conditioned floor area by to get the effective mass area by building construction type. This is used for the calculation of the effective mass area "Am" in *get_prop_RC_model*. Standard values can be found in the Annex G of ISO EN13790

param cm: The internal heat capacity per unit of area [J/m2].

return Effective mass area factor (0, 2.5 or 3.2 depending on cm value).

omegalpes.energy.buildings.thermal.**write_linerazation_exp**(*T_dry*, *T_dew*, *sky_cover*, *Tlin*, *U_win*, *U_wall*, *U_roof*, *e_win*, *e_wall*, *e_roof*, *A_win*, *A_wall*, *A_roof*, *name*)

Exergy module

This module contains the exergy assessment routines of OMEGALPES. This module:

1) Determines inlet, outlet or contained exergy of, respectively, any ConsumptionUnit, ProductionUnit or Storage-Unit.

2) Determines exergy destruction within any EnergyUnit. 2) Recognizes Electrical and Thermal energy. 3) Can calculate exergy for a single unit or for a list of units. 4) Can proceed with only one temperature value or with a list of temperatures.

4.1. Formulates exergy for one single EnergyUnit and temperature. 4.2. Formulates timely exergy if one single EnergyUnit and a list of temperatures is provided. 4.3. Formulates exergy for each unit within a list of EnergyUnits if only

one temperature is provided.

4.4. Formulates timely exergy for each unit within a list of EnergyUnits if a list of temperatures is provided.

The exergy-related classes defined in this module are not physical units. They are virtual units attached to their energetic counterparts. Consequently, the exergy and exergy destruction calculated in this module are attached to the EnergyUnit as a Quantity at the moment of calculating it.

class omegalpes.energy.exergy.**ElectricalExergy**(*energy_unit: omegalpes.energy.units.energy_units.EnergyUnit*)

Bases: *omegalpes.general.optimisation.core.OptObject*

class omegalpes.energy.exergy.**ExergyDestruction**(*energy_unit=None*, *exergy_eff=1*, *temp_ref=20*, *temp_heat=None*)

Bases: *omegalpes.general.optimisation.core.OptObject*

class omegalpes.energy.exergy.**ThermalExergy**(*energy_unit: omegalpes.energy.units.energy_units.EnergyUnit*, *temp_heat: int*, *temp_ref=20*)

Bases: *omegalpes.general.optimisation.core.OptObject*

Poles module

This module defines inputs and outputs of as poles

The poles module includes :

- FlowPole : this class defines a pole with a directed flow (in or out)
- EPole : this class define an energy pole

class omegalpes.energy.io.poles.**Epole**(*p*, *direction*, *energy_type=None*)
Bases: *omegalpes.energy.io.poles.FlowPole*

Description

Definition of an energetic pole, power and power flow direction convention 'in' or 'out'

class omegalpes.energy.io.poles.**FlowPole**(*flow='flow'*, *direction='in'*)
Bases: dict

Description

Interface for basics flux poles

The energy package also includes an exergy module

2.3.2 exergy module

The exergy module functionalities can be found in the energy package.

The exergy package serves the primary purpose of determining exergy destruction within a unit, and declaring it as an attribute (Quantity) of the unit, eligible as objective function for the optimizations. The package uses three main modules as pillars: 'ElectricalExergy', 'ThermalExergy' and 'ExergyDestruction'. The first two identify, respectively, the electrical or thermal flows interacting with an energy unit, and determine the corresponding exergy flow accordingly. Then, the "ExergyDestruction" module reads the exergy flows assessed for the unit, and applies an exergy balance to determine its exergy destruction.

The module 'ElectricalExergy' determines the exergy flow associated to an input, output or accumulation of electricity in a consumption, production or storage unit respectively. 'ElectricalExergy' starts by verifying that the unit is indeed an electrical unit (energy_type = 'elec'), raising a 'TypeError' otherwise. After this verification, it calculates the exergy flow and declares it as an attribute of the unit (energy_unit.exergy). The calculation is rather straightforward, as in the case of electricity, exergy equals energy.

The module 'ThermalExergy' determines the exergy flow associated to an input, output or accumulation of heat in a consumption, production or storage unit respectively. 'ThermalExergy' starts by verifying that the unit is indeed a thermal unit (energy_type = 'thermal'), raising a 'TypeError' otherwise. After this verification, it calculates the exergy flow and declares it as an attribute of the unit (energy_unit.exergy). Then, it calculates thermal exergy based on the following general formulas (first one for production/consumption units, and second one for storage units):

{0}_exergy[t] == {0}_p[t]*(1 -({1}+273.15)/({1}+273.15))*time.DT

{0}_exergy[t] == {0}_e[t]*(1 -({1}+273.15)/({0}_t_heat+273.15))*time.DT'

with {0} the energy unit name, and {1} the dead state temperature.

The procedure is accompanied with several checks on correct values inputted for temperature.

Before carrying out their calculations, both 'ElectricalExergy' and 'ThermalEnergy' verify that the unit is indeed an energy unit, that exergy has not yet been assessed for it, and that its energy type corresponds to the target type. This is enforced through the methods named _check_energy_unit and _check_exergy_not_assessed.

The module named 'ExergyDestruction' allows determining exergy destruction within a unit, and enabling that magnitude as an optimization objective. This module requires as input the name of the target energy unit ('energy_unit'), its exergy efficiency ('exergy_eff'), the dead state temperature for exergy analysis ('temp_ref'), and in the case of thermal storage units, their temperature level ('temp_heat'). It also requires that the exergy flow of the unit have been determined preemptively, by the mean of either "ElectricalExergy" or "ThermalExergy". The inputs named 'energy_unit' and 'temp_heat' default to 'None', since it is imperative that the user specifies them. Meanwhile, 'exergy_eff' defaults to 1, meaning that the module assumes perfect efficiency unless otherwise stated. The dead state temperature

('temp_ref') defaults to 20 °C, as in the case of the "ThermalExergy" module. Users can change this value, but must be aware that all units in the study must have the same 'temp_ref', for the sake of thermodynamic consistency.

First, 'ExergyDestruction' verifies that exergy destruction has not been already assessed for that unit, raising 'AttributeError' otherwise. Then, it declares three new attributes for 'energy_unit'. The first two are the 'Quantities' named 'exergy_dest' and 'exd_tot', which represent exergy destroyed at each time step and throughout the whole timespan, respectively. The third attribute is a 'DefinitionConstraint' named 'calc_exd_tot' that ensures calculation of 'exd_tot', whose formulation is the same in all cases. Meanwhile, calculation of 'exergy_dest' is dynamic and has different formulations depending on the type of unit. Thus, this calculation is declared later in subsequent routines.

If the unit is a 'ConsumptionUnit', the module first checks that the user has provided a value for 'exergy_eff', and raises 'ValueError' otherwise. Then, it proceeds to the formulation of a 'DefinitionDynamicConstraint' named 'calc_exergy_dest', which enables calculation of 'exergy_dest' as mentioned above. For a consumption unit, exergy destruction is the product of its input exergy by the difference between 1 and its exergy efficiency. The equation has two formulations, one adapted for 'exergy_eff' being an int or float, and the other for it being a list. If it is none of those types, the module raises 'TypeError'. Additionally, if it is a list the module verifies that its length matches that of the 'time' list, raising 'IndexError' otherwise. It also verifies that each element of the list is either an int or a float, raising 'ValueError' otherwise.

If the unit is a 'ProductionUnit', the module applies the same routines as for a 'ConsumptionUnit', with just two slight variations. The main variation occurs in the calculation of exergy destruction. For a production unit, exergy destruction is assessed as the product between its exergy output and the difference between the inverse of its efficiency and 1. This calculation means that a production unit is actually a conversion unit whose input is not necessary for energy modelling, but it is for exergy modelling. Lower exergy efficiency means that the production unit has needed a greater input of primary energy for delivering energy. The second variation stems from the first; the module needs to check that the user has entered values greater than zero for 'exergy_eff'. Indeed, a production unit cannot produce if its efficiency is zero.

If the unit is a 'StorageUnit', its exergy destruction is directly related to its energy losses. Thus, the module multiplies those losses by the corresponding exergy factor. That factor equals one for electricity, while for heat it is a function of temperature. Thermal storage units are the only context where the "ExergyDestruction" module needs an input for 'temp_heat' and 'temp_ref'.

If the unit is a 'ConversionUnit', the module reads the lists of production and consumption sub-units enclosed by the unit. For each sub-unit, it first verifies that its 'exergy' has been assessed, raising 'AttributeError' otherwise. Then, the module includes the sub-unit in the exergy balance of the overall unit. It does so by writing its 'exergy' attribute within the equation, preceded by a plus sign in the case of consumption sub-units, or a minus sign in the case of production sub-units. Once the full equation is ready, the module declares it as 'calc_exergy_dest'.

If the unit provided to 'ExergyDestruction' is not a consumption, production, storage or conversion unit, the module returns 'TypeError'. The implementation of newest unit types in the module might be in the developers' portfolio in the future.

With both exergy and exergy destruction being assessed, the purpose of the exergy package is fulfilled.

Users should be aware of a few methodological simplifications within the exergy package. Firstly, it cannot assess the following forms of exergy: chemical, potential and kinetic. If the user aims at properly studying such types of energy, other tools are available elsewhere. Nevertheless, if units managing those sources are not the center of the study, it is theoretically possible to "pseudo-model" them. Users can declare them as thermal units, and adjust its temperature level so that the resulting exergy factor is equivalent to that of the chemical, potential or kinetic energy flow. Finding that equivalence requires advanced knowledge in exergy analysis; let users adopt this approach under their own responsibility. The proper modeling of chemical, potential or kinetic exergy might be in the developers' portfolio in the future.

A second simplification concerns the assessment of thermal exergy. Namely, the exergy of transferred heat depends strongly on temperature levels. Despite that, the exergy module assesses thermal exergy assuming that temperature levels remain constant. If that is not true, deviations appear with respect to the most accurate assessment (based on physical exergy of the fluid carrier). Those deviations become larger as temperature changes in the carriers become

larger. If a heat source undergoes a large temperature drop throughout the exchange of heat, then its exergy content may be largely overestimated. Moreover, deviations are amplified if temperature levels are close to the dead state temperature. In such instances, the developers recommend using the average of inlet-outlet temperatures of the source, instead of the inlet temperature. The dead state temperature typically receives the symbol T0 in exergy analyses, and was named temp_ref within the exergy package. Users with advanced knowledge on exergy can adjust the value of temperature so that deviations become minimal with respect to the most accurate assessment.

Lastly, users should be aware that the term "ExergyDestruction" assessed within the module aggregates two concepts: losses of energy (and thus of exergy) from the unit to its surroundings, and thermodynamic irreversibility within the unit itself. Experts in exergy analysis typically call the former 'exergy losses', the latter 'exergy destruction', and 'irreversibility' the aggregate thereof. Regardless, the exergy module assesses only the aggregate. This simplification is acceptable; energy planners focus mainly on overall efficiency, and do not need the details on the sources of irreversibility. The term "ExergyDestruction" determined in this module suffices for assessing exergy efficiency of units and systems. Besides, the module names that magnitude "ex_dest", for "exergy destruction", instead of 'irreversibility'. The developers intentionally avoided using "Irreversibility" as class or variable name in this module. This was done to anticipate possible conflicts in nomenclature with respect to reversible energy units ("ReversibleUnit" class), which were under development at the time.

The **actor classes** enables to build the energy model considering pre-defined stakeholders' constraints and objectives

2.3.3 actor package

OMEGAlpes offers a library of models associated with the "multi-actor" approach through the Actor package. An explicit actor model aims to differentiate technical issues from stakeholder-induced issues, and to differentiate the decisions made by the stakeholders. This model does not attempt to model the actors as such but is restricted to the constraints and objectives carried by the actors that influence the project framework.

It is a matter of being able to differentiate the technical part of the model from the part associated with the actors. When modelling, this approach can lead stakeholders to question what is technical and what is related to actor's choices. It also helps to provide additional insights in stakeholder negotiations by identifying whether the issues are technical or actor-related. Negotiation can then be facilitated by the fact that decisions are associated with their stakeholders and their influence on the project can be assessed.

The modeling logic behind actor package is debatable, especially as, by refining the technical considerations, it is possible to take it into account on the energy layer. However, we are convinced that bringing stakeholders into the modelling loop can facilitate the technical refinement of the energy model and can help decision-making and negotiations between stakeholders.

The actor modelling is based on a main actor class defined on the actor module. Then, the actors are divided in two categories: the "operator actors" who operates energy units and the "regulator actors" who unable to create regulation constraints.

Actor module

This modules define the basic Actor object

Few methods are available:

- add_external_constraint
- add_external_dynamic_constraint
- add_objective

class omegalpes.actor.actor.**Actor**(*name*, *no_warn=True*, *verbose=True*)
Bases: *omegalpes.general.optimisation.core.OptObject*

Description

Actor class is the basic class to model an actor. The basic actor is defined by its name and description.
An actor is then defined by its constraints and objectives.

Attributes

- description : description as an Actor OptObject

add_actor_constraint(*cst_name*, *exp*)
Enable to add an external constraint linked with an actor

Parameters

- **cst_name** – name of the constraint
- **exp** – expression of the constraint

add_actor_dynamic_constraint(*cst_name*, *exp_t*, *t_range='for t in time.I'*)
Enable to add an external dynamic constraint linked with an actor. A dynamic constraint changes over time

Parameters

- **cst_name** – name of the constraint
- **exp** – expression of the constraint depending on the time
- **t_range** – expression of time for the constraint

add_objective(*obj_name*, *exp*)
Enable to add an objective linked with an actor

Parameters

- **obj_name** – name of the objective
- **exp** – expression of the objective

remove_actor_constraints(*ext_cst_name_list=None*)
Enable to remove an external constraint linked with an actor

Parameters **ext_cst_name_list** – list of external constraint that would be removed

The Actor class is defined as an abstract class, i.e. not specific to a particular actor, and has generic methods for adding or removing constraints and goals. Regarding the constraints, a distinction is made between:

- definition constraints (DefinitionConstraint), used to calculate quantities or to represent physical phenomena (e.g. storage state of charge) considered a priori as non-negotiable;
- technical constraints (TechnicalConstraint), used to represent technical issues (e.g. operating power of a production unit) considered a priori as non-negotiable unless technical changes are made;

- actor-specific constraints, ActorConstraint and ActorDynamicConstraint, to model a constraint that is due to actor decisions (e.g. minimal level of energy consumption over a period).

Those constraints are exclusive, and only actor-specific constraints are a priori negotiable as decided by the stakeholders themselves. Actors modeling includes additional objectives. In OMEGAlpes, a weight can be added to an objective in order to give more importance to one or more objectives compared to the others.

Area of responsibility

In order to link the energy units to the actors, an area of responsibility is defined for each actor in the model. These are the energy units they operate (for the operating stakeholders) or build (for the developer). From a modelling point of view, this notion of area of responsibility limits the constraints and objectives of the stakeholders, which can only be applied to the area of responsibility of the latter. To do so , it is necessary to define the stakeholder's responsibility space in the modelling by using two attributes associated to the Actor class:

- operated_unit_list : to indicate the energy units within the stakeholder's area of responsibility,
- operated_node_list : to indicate which nodes are part of the stakeholder's area of responsibility.

Operator_actors module

This module defines the operator_actor and its scope of responsibility

class omegalpes.actor.operator_actors.operator_actors.**OperatorActor**(*name*, *operated_unit_type_tuple*, *operated_unit_list=None*, *operated_node_list=None*, *verbose=True*)

Bases: *omegalpes.actor.actor.Actor*

Description

OperatorActor class inherits from the the basic class Actor. It enables one to model an actor who operates energy units which is part of its scope of responsibility. An operator actor has objectives and constraints which are linked to the energy units he operates.

Attributes

- name : name of the actor
- operated_unit_list: list of the energy units operated by the actor or more precisely in its scope of responsibility

Operators can only influence - with respect to constraints and objectives - the units within its scope of responsibility, as defined in the following sub-section. Based on a typology of operator actors, we have developed the following classes: Consumer, Producer, Prosumer, Supplier.

Consumer_actors module

This module describes the Consumer actor

Few objectives and constraints are available.

Objectives :

- maximize_consumption
- minimize_consumption
- minimize_co2_consumption
- minimize_consumption_costs

Constraints :

- energy_consumption_minimum
- energy_consumption_maximum
- power_consumption_minimum
- power_consumption_maximum

class omegalpes.actor.operator_actors.consumer_actors.**Consumer**(*name*, *operated_unit_list*, *operated_node_list=None*, *verbose=True*)

Bases: *omegalpes.actor.operator_actors.operator_actors.OperatorActor*

Description

Consumer class inherits from the the class OperatorActor. It enables one to model a consumer actor.

add_energy_consumption_maximum(*max_e_tot*, *cst_operated_unit_list=None*)

To create the actor constraint of a maximum of energy consumption.

Parameters

- **max_e_tot** – Maximum of the total energy consumption over the study period
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_energy_consumption_minimum(*min_e_tot*, *cst_operated_unit_list=None*)

To create the actor constraint of a minimum of energy consumption.

Parameters

- **min_e_tot** – Minimum of the total energy consumption over the study period
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_power_consumption_by_unit_maximum(*max_p*, *time*, *cst_operated_unit_list=None*)

To create the actor constraint of a maximum of power consumption for each unit.

Parameters

- **max_p** – Maximum of the power consumption. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_power_consumption_by_unit_minimum(*min_p*, *time*, *cst_operated_unit_list=None*)

To create the actor constraint of a minimum of power consumption for each unit.

Parameters

- **min_p** – Minimum of the power consumption. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_power_consumption_total_maximum(*max_p*, *time*, *cst_operated_unit_list=None*)
To create the actor constraint of a maximum of power consumption.

Parameters

- **max_p** – Minimum of the power consumption. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_power_consumption_total_minimum(*min_p*, *time*, *cst_operated_unit_list=None*)
To create the constraint of a minimum of power consumption considering all the units.

Parameters

- **min_p** – Minimum of the power consumption. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

maximize_consumption(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)
To create the objective in order to maximize the consumption of the consumer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – List of units on which the objective will be applied. Might be empty
- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_co2_consumption(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)
To create the objective in order to minimize the co2 emissions due to the consumer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – List of units on which the objective will be applied. Might be empty
- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_consumption(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)
To create the objective in order to minimize the consumption of the consumer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – List of units on which the objective will be applied. Might be empty
- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_consumption_cost(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)

To create the objective in order to minimize the expenses due to the consumer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – List of units on which the objective will be applied. Might be empty
- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

Consumer class inherits from the the class OperatorActor. It enables one to model a consumer actor.

Producer_actors module

This module describes the Producer actor

Few objectives and constraints are available.

Objectives :

- maximize_production
- minimize_production
- minimize_time_of_use
- minimize_co2_emissions
- minimize_costs
- minimize_operating_cost
- minimize_starting_cost

Constraints :

- energy_production_minimum
- energy_production_maximum
- power_production_minimum
- power_production_maximum

class omegalpes.actor.operator_actors.producer_actors.**Producer**(*name*, *operated_unit_list*, *operated_node_list=None*, *verbose=True*)

Bases: *omegalpes.actor.operator_actors.operator_actors.OperatorActor*

Description

Producer class inherits from the the class OperatorActor. It enables one to model an energy producer actor.

add_energy_production_maximum(*max_e_tot*, *cst_operated_unit_list=None*)
To create the actor constraint of a maximum of energy production.

Parameters

- **max_e_tot** – Maximum of the total energy production over the period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_energy_production_minimum(*min_e_tot*, *cst_operated_unit_list=None*)
To create the actor constraint of a minimum of energy production.

Parameters

- **min_e_tot** – Minimum of the total energy production over the period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_power_production_by_unit_maximum(*max_p*, *time*, *cst_operated_unit_list=None*)
To create the actor constraint of a maximum of power production for each unit.

Parameters

- **max_p** – Maximum of the power production. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_power_production_by_unit_minimum(*min_p*, *time*, *cst_operated_unit_list=None*)
To create the actor constraint of a minimum of power production for each unit.

Parameters

- **min_p** – Minimum of the power production. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_power_production_total_maximum(*max_p*, *time*, *cst_operated_unit_list=None*)
To create the actor constraint of a maximum of power production.

Parameters

- **max_p** – Minimum of the power production. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_power_production_total_minimum(*min_p*, *time*, *cst_operated_unit_list=None*)
To create the constraint of a minimum of power production considering all the units.

Parameters

- **min_p** – Minimum of the power production. May be an int, float or a list with the size of the period study

- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

add_temporary_stop(*time*, *start='YYYY-MM-DD HH:MM:SS'*, *end='YYYY-MM-DD HH:MM:SS'*, *period_index=None*, *cst_production_unit_list=None*)

To create the actor constraint giving the possibility to stop the production during a period.

Parameters

- **start** – start of the stop period
- **end** – end of the stop period
- **period_index** – period index for the stop period, to use instead of start and end
- **cst_production_unit_list** – List of units on which the constraint will be applied. Might be empty.

maximize_production(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)

To create the objective in order to maximize the production of the producer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – List of units on which the objective will be applied. Might be empty.
- **weight** – Weight coefficient for the objective

minimize_co2_emissions(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)

To create the objective in order to minimize the co2 emissions of the producer's units (all or part of them). based on the quantity "co2_emission"

Parameters

- **obj_operated_unit_list** – list of the operated energy units on which the objective will be applied
- **weight** – weight of the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_costs(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)

To create the objective in order to minimize the cost of the producer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – list of the operated energy units on which the objective will be applied
- **weight** – weight of the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_operating_cost(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)

To create the objective in order to minimize the operating costs of the producer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – list of the operated energy units on which the objective will be applied
- **weight** – weight of the objective

- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_production(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)

To create the objective in order to minimize the production of the producer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – List of units on which the objective will be applied. Might be empty.
- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_starting_cost(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)

To create the objective in order to minimize the starting costs of the producer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – list of the operated energy units on which the objective will be applied
- **weight** – weight of the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

minimize_time_of_use(*obj_operated_unit_list=None*, *weight=1*, *pareto=False*)

To create the objective in order to minimize the time of use of the producer's units (all or part of them).

Parameters

- **obj_operated_unit_list** – List of units on which the objective will be applied. Might be empty.
- **weight** – Weight coefficient for the objective
- **pareto** – if True, OMEGAlpes calculates a pareto front based on this objective (two objectives needed)

power_consumption_maximum(*max_p*, *time*, *cst_operated_unit_list=None*)

DEPRECATED: please use

add_power_production_total_maximum or

add_power_production_by_units_maximum instead

To create the actor constraint of a maximum of power consumption.

Parameters

- **max_p** – Maximum of the power consumption. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

power_consumption_minimum(*min_p*, *time*, *cst_operated_unit_list=None*)

DEPRECATED: please use

add_power_production_total_minimum or

add_power_production_by_units_minimum instead

To create the actor constraint of a minimum of power consumption.

Parameters

- **min_p** – Minimum of the power consumption. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_operated_unit_list** – List of units on which the constraint will be applied. Might be empty.

Producer class inherits from the the class OperatorActor. It enables one to model an energy producer actor.

Prosumer_actors module

Prosumer class inherits from the the class OperatorActor, Consumer and Producer. It enables one to model an actor which is at the same time an energy producer and consumer

Regulator_actors module

This module defines the operator_actor and its scope of responsibility

One constraint is available :

- co2_emission_maximum

class omegalpes.actor.regulator_actors.regulator_actors.**LocalAuthorities**(*name*)
Bases: *omegalpes.actor.regulator_actors.regulator_actors.RegulatorActor*

Description

LocalAuthorities class inherits from the basic class RegulatorActor. It focuses on local Authorities constraints

class omegalpes.actor.regulator_actors.regulator_actors.**RegulatorActor**(*name*, *verbose=True*)
Bases: *omegalpes.actor.actor.Actor*

Description

RegulatorActor class inherits from the the basic class Actor. It enables one to model an actor who can add constraints on all energy units of the study case.

Attributes

- name : name of the actor

add_co2_emission_maximum(*time*, *co2_max*, *cst_production_list*)
To create the actor constraint of a maximum of CO2 emission.

Parameters

- **co2_max** – Maximum of the CO2 emission. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_production_list** – List of production units on which the

constraint will be applied.

add_particles_emission_maximum(*time*, *particle_max*, *cst_production_list*)
To create the actor constraint of a maximum of CO2 emission.

Parameters

- **particle_max** – Maximum of the particle emission. May be an int, float or a list with the size of the period study
- **time** – period of the study
- **cst_production_list** – List of production units on which the

constraint will be applied.

class omegalpes.actor.regulator_actors.regulator_actors.**StateAuthorities**(*name*)
Bases: *omegalpes.actor.regulator_actors.regulator_actors.RegulatorActor*

Description

StateAuthorities class inherits from the basic class RegulatorActor. It focuses on local Authorities constraints

Regulators do not operate any particular energy unit but influence the energy system with regard to grid and/or resource regulation. Their decisions can affect all energy units. From a modelling point of view, we have modelled the RegulatorActor class which inherits from the Actor class.

The regulatory actors are divided into two main categories: local regulatory authorities (e.g. local authorities) and regulatory authorities outside the project (e.g. State, National Energy Regulator entities, such as CRE in France), respectively modelled by the classes LocalAuthority and StateAuthority.

The **general classes** helps to build the units, the model and to plot the results

2.3.4 general package

Please, have a look to the following general modules

Firstly, the main modules to build an energy optimisation model are presented :

Elements module

This module includes the optimization elements (quantities, constraints and objectives) formulated in LP or MILP

- Quantity : related to the decision variable or parameter

- Constraint : related to the optimization problem constraints 4 categories of constraints :
  - Constraint: to define a constraint object
  - DefinitionConstraint: to calculate and define quantities or for physical

  constraints - TechnicalConstraint: linked with technical constraints - ActorConstraint: constraint decided by the actors 2 types of constraints: - Constraints : basic constraint - DynamicConstraint: basic constraint depending on the time
- Objective : related to the objective function

class omegalpes.general.optimisation.elements.**ActorConstraint**(*exp*, *name='ActCST0'*, *description=''*, *active=True*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.Constraint*

Description

Defining a category of constraint: ActorConstraint (see: DynamicConstraint, TechnicalConstraint) To model a constraint which is due to actor decisions. Must be different from Definition and Technical constraints. This kind of constraint is mainly a negotiable constraint.

example: to have a minimal level of energy consumption over a period. Constraint : min_energy

deactivate_constraint()

An actor's constraint can be deactivated : - To compare scenarios - To try a less constrained problem

class omegalpes.general.optimisation.elements.**ActorDynamicConstraint**(*exp_t*, *t_range='for t in time.I'*, *name='ActDynCST0'*, *active=True*, *description='Actor and dynamic constraint'*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.DynamicConstraint*, *omegalpes.general.optimisation.elements.ActorConstraint*

Description

Defining a category of constraint: ActorDynamicConstraint (see: DynamicConstraint, ActorConstraint) Must be different from Definition and Technical constraints. This kind of constraint is mainly a negotiable constraint.

example: to operate an energy unit only on defined periods. Constraint: daily_dynamic_constraint

class omegalpes.general.optimisation.elements.**Constraint**(*exp*, *name='CST0'*, *description=''*, *active=True*, *parent=None*)

Bases: object

Description

Class that defines a constraint object

Attributes

- name: name of the constraint
- description: a description of the constraint
- active: False = non-active constraint; True = active constraint
- exp: (str) : expression of the constraint
- parent: (unit) : this constraint belongs to this unit

Note: Make sure that all the modifications on Constraints are made before adding the unit to the Model (OptimisationModel.addUnit()).

---

class omegalpes.general.optimisation.elements.**DailyDynamicConstraint**(*exp_t*, *time*, *init_time: str = '00:00'*, *final_time: str = '23:00'*, *name='daily_dynamic_constrai*
description='daily dynamic constraint', *active=True*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.ActorDynamicConstraint*

Description

Class that defines a daily dynamic constraint for a time range

Ex: Constraint applying between 7am and 10:30pm

ex_cst = DailyDynamicConstraint(exp_t, time, init_h="7:00", final_h="10:30", name='ex_cst')

Attributes

- name (str) : name of the constraint
- exp_t (str) : expression of the constraint

- init_time (str) : starting time of the constraint in the format:

"HH:MM", consistent with the dt value. - final_time (str) : ending time of the constraint in the format: "HH:MM", consistent with the dt value. - description (str) : description of the constraint - active (bool) : defines if the constraint is active or not - parent (OptObject) : parent of the constraint

class omegalpes.general.optimisation.elements.**DefinitionConstraint**(*exp*, *name='DefCST0'*, *description='Constraint for definitions'*, *active=True*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.Constraint*

Description

Defining a category of constraint: DefinitionConstraint for imposed mathematical relation constraints, which may be physical. They are used to calculate and define quantities, or to represent physical phenomenon. Must be different from Technical and Actor constraints. This kind of constraint is by nature non-negotiable

Definition constraint: added to the model to calculate a quantity or define it considering an other.

example: the definition of e_tot: calc_e_tot (e_tot = LpSum(e))

Physical constraint: to model a constraint which is linked to the physics.

example: not implemented yet, see DefinitionDynamicConstraint

class omegalpes.general.optimisation.elements.**DefinitionDynamicConstraint**(*exp_t*, *t_range='for t in time.I'*, *name='DefDynCST0'*, *description='dynamic constraint for definition'*, *active=True*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.DynamicConstraint*, *omegalpes.general.optimisation.elements.DefinitionConstraint*

Description

Defining a category of constraint: DefinitionDynamicConstraint on the time. (see: DynamicConstraint, DefinitionConstraint) Must be different from Technical and Actor constraints. This kind of constraint is by nature non-negotiable

DefinitionDynamicConstraint: to calculate a quantity or define it considering an other quantity and depending on the time.

example: the definition of a capacity def_capacity (energy[t] <= capacity at each time step t)

Physical constraint: to model physical constraints.

example: the power balance in an Energy node: power_balance

(LpSum(p_production_unit[t] for the set of production units connected to the energy node) = LpSum(p_consumption_unit[t] for the consumption units connected to the energy node at each time step t)

class omegalpes.general.optimisation.elements.**DynamicConstraint**(*exp_t*, *t_range='for t in time.I'*, *name='DCST0'*, *description='dynamic constraint'*, *active=True*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.Constraint*

Description

Class that defines a constraint depending on the time. NB : Mandatory for PuLP

class omegalpes.general.optimisation.elements.**ExtDynConstraint**(*exp_t*, *t_range='for t in time.I'*, *name='EDCST0'*, *active=True*, *description='Non-physical and dynamic constraint'*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.DynamicConstraint*, *omegalpes.general.optimisation.elements.ExternalConstraint*

Description

DEPRECATED: please use ActorDynamicConstraint

Defining a constraint both external and dynamic (see: DynamicConstraint, ExternalConstraint)

class omegalpes.general.optimisation.elements.**ExternalConstraint**(*exp*, *name='ExCST0'*, *description=''*, *active=True*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.Constraint*

Description

DEPRECATED: please use ActorConstraint

Defining a special type of constraint: the external constraint

- This constraint does not translate a physical constraint
- This constraint defines an external constraint, which could be relaxed

deactivate_constraint()
An external constraint can be deactivated : - To compare scenarios - To try a less constrained problem

class omegalpes.general.optimisation.elements.**HourlyDynamicConstraint**(*exp_t*, *time*, *init_h: int = 0*, *final_h: int = 24*, *name='HDCST0'*, *description='hourly dynamic constraint'*, *active=True*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.ActorDynamicConstraint*

Description

DEPRECATED: please use DailyDynamicConstraint

Class that defines an dynamic contraint for a time range

Ex: Constraint applying between 7am and 10pm

ex_cst = HourlyDynamicConstraint(exp_t, time, init_h=7, final_h=22, name='ex_cst')

Attributes

- name (str) : name of the constraint
- exp_t (str) : expression of the constraint
- init_h (int) : hour of beginning of the constraint [0-23]
- final_h (int) : hour of end of the constraint [1-24]
- description (str) : description of the constraint
- active (bool) : defines if the constraint is active or not
- parent (OptObject) : parent of the constraint

class omegalpes.general.optimisation.elements.**Objective**(*exp*, *name='OBJ0'*, *description=''*, *active=True*, *weight=1*, *unit='s.u.'*, *pareto=False*, *parent=None*)

Bases: object

Description

Class that defines an optimisation objective

Attributes

- name (str) :
- exp (str) :
- description (str) :
- active (bool) :
- weight (float) : weighted factor of the objective
- parent (unit)
- unit (str) : unit of the cost expression
- **pareto (str)** [if True, OMEGAlpes calculates a pareto front based on] this objective (two objectives needed)

Methods

- _add_objectives_list()
- _add_pareto_objectives_list()

Note: Make sure that all the modifications on Objectives are made before adding the unit to the OptimisationModel, otherwise, it won't be taken into account

---

class omegalpes.general.optimisation.elements.**Quantity**(*name='var0'*, *opt=True*, *unit='s.u'*, *vlen=None*, *value=None*, *description=''*, *vtype='Continuous'*, *lb=None*, *ub=None*, *parent=None*)

Bases: object

Description

Class that defines what is a quantity. A quantity can wether be a decision variable or a parameter, depending on the opt parameter

Attributes

- name (str) : the name of the quantity
- description (str) : a description of the meaning of the quantity
- vtype (PuLP) : the variable type, depending on PuLP
  - LpBinary (binary variable)
  - LpInteger (integer variable)
  - LpContinuous (continuous variable)
- vlen (int) : size of the variable
- unit (str) : unit of the quantity
- opt (binary)

  - True: this is an optimization variable
  - False: this is a constant - a parameter
- value (float, list, dict) : value (unused if opt=True)
- ub, lb : upper and lower bounds
- parent (OptObject) : the quantity belongs to this unit

Note: Make sure that all the modifications on Quantity are made before adding the unit to the Model

---

get_value()
return the value of the quantity according the type of the value in order to be able to use it easily in print and plot methods: int -> int float -> float list -> list dict -> list ndarray –> list

get_value_with_date(*unit=None*)
return the values of the quantity associated to a date in a dataframe if the values are a list or a dict (only).

class omegalpes.general.optimisation.elements.**TechnicalConstraint**(*exp*, *name='TechCST0'*, *description='Technical constraint'*, *active=True*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.Constraint*

Description

Defining a category of constraint: TechnicalConstraint. To model a constraint which is linked to technical issues Must be different from Definition and Actor constraints. This kind of constraint is mainly a negotiable constraint.

example: not implemented yet, see TechnicalDynamicConstraints

deactivate_constraint()
An technical constraint can be deactivated : - To compare scenarios - To try a less constrained problem

class omegalpes.general.optimisation.elements.**TechnicalDynamicConstraint**(*exp_t*, *t_range='for t in time.I'*, *name='TechCST0'*, *active=True*, *description='Technical and dynamic constraint'*, *parent=None*)

Bases: *omegalpes.general.optimisation.elements.DynamicConstraint*, *omegalpes.general.optimisation.elements.TechnicalConstraint*

Description

Defining a category of constraint: TechnicalDynamicConstraint depending on the time. (see: DynamicConstraint, TechnicalConstraint) Must be different from Definition and Actor constraints. This kind of constraint is mainly a negotiable constraint.

example: an energy unit may not be able to stop in one time step, but several. The power decreases as a ramp shape: set_max_ramp_down

Core module

This module define the main Core object on which the energy units and actors will be based

class omegalpes.general.optimisation.core.**OptObject**(*name='U0'*, *description='Optimization object'*, *verbose=True*)

Bases: object

Description

OptObject class is used as an "abstract class", i.e. it defines some general attributes and methods but doesn't contain variable, constraint nor objective. In the OMEGAlpes package, all the subsystem models are represented by a unit. A model is then generated adding OptObject to it. Variable, objective and constraints declarations are usually done using the __init__ method of the OptObject class.

Attributes

- name
- description
- _quantities_list : list of the quanitities of the OptObject (active or not)
- _constraints_list : list of the constraints of the OptObject(active or not)
- _technical_constraints_list : list of the constraints of the OptObject (active or not)
- _objectives_list : list of the objectives of the OptObject (active or not)

Methods

- __str__: defines the
- __repr__: defines the unit with its name
- get_constraints_list
- get_constraints_name_list
- get_external_constraints_list
- get_external_constraints_name_list
- get_objectives_list
- get_objectives_name_list
- get_quantities_list
- get_quantities_name_list
- deactivate_optobject_external_constraints

Note: The OptObject class shouldn't be instantiated in a python script, except if you want to create your own model from the beginning. In this case, one should consider creating a new class NewModel(OptObject).

deactivate_optobject_external_constraints(*ext_cst_name_list=None*)
Enable to remove an external constraint linked with an OptObject

Parameters **ext_cst_name_list** – list of external constraint that would be removed

get_actor_constraints_list()
Get the technical constraints associated with the unit as a dictionary shape ['constraint_name' , constraint]

get_actor_constraints_name_list()
Get the names of the external constraints associated with the unit

get_constraints_list()
Get the constraints associated with the unit as a dictionary shape ['constraint_name' , constraint]

get_constraints_name_list()
Get the names of the constraints associated with the unit

get_objectives_list()
Get objectives associated with the unit as a dictionary shape ['objective_name' , objective]

get_objectives_name_list()
Get the names of the objectives associated with the unit

get_quantities_list()
Get the quantities associated with the unit as a dictionary shape ['quantity_name' , quantity]

get_quantities_name_list()
Get the names of the quantities associated with the unit

get_technical_constraints_list()
Get the technical constraints associated with the unit as a dictionary shape ['constraint_name' , constraint]

get_technical_constraints_name_list()
Get the names of the external constraints associated with the unit

Time module

This module creates the Time object

class omegalpes.general.time.**TimeUnit**(*start='01/01/2018'*, *end=None*, *periods=None*, *dt=1*, *verbose=True*)

Bases: object

Description

Class defining the studied time period.

Attributes

- DATES : dated list of simulation steps
- DT : delta t between values in hours (int or float), i.e. 1/6 will be 10 minutes.
- LEN : number of simulation steps (length of DATES)
- I : index of time ([0 : LEN])

get_date_for_index(*index*)
Getting a date for a given index

Parameters **index** – int value for the index of the wanted dated, between 0 and LEN (it must be in the studied period)

get_days
Getting days for the studied period

Return all_days list of days of the studied period

get_index_for_date(*date='YYYY-MM-DD HH:MM:SS'*)
Getting the index associated with a date

Parameters **date** – date the index of is wanted. Format YYYY-MM-DD HH:MM:SS, must be within the studied period and consistent with the timestep value

get_index_for_date_range(*starting_date='YYYY-MM-DD HH:MM:SS'*, *end=None*, *periods=None*)
Getting a list of index for a date range

Parameters

- **starting_date** – starting date of the wanted index
- **end** – ending date of the wanted index
- **periods** – number of periods from the starting_date of the wanted index

Return index_list list of indexes for the given dates

get_non_working_dates(*month_range=range(0, 12)*, *hour_range=range(0, 24)*, *country='France'*)

get_non_working_days(*country='France'*)

get_working_dates(*month_range=range(0, 12)*, *hour_range=range(0, 24)*, *country='France'*)

get_working_days(*country='France'*)

print_studied_period()

omegalpes.general.time.**convert_european_format**(*date*)

Converting a date with an european format DD/MM/YYYY into a datetime format YYYY-MM-DD or return

Parameters **date** – date in european format

Returns date in format datetime

Model module

This module enables to fill the optimization model and formulate it in LP or MILP based on the package PuLP (LpProblem)

class omegalpes.general.optimisation.model.**OptimisationModel**(*time*, *name='optimisation_model'*)

Bases: pulp.pulp.LpProblem

Description

This class includes the optimization model formulated in LP or MILP based on the package PuLP (LpProblem)

add_nodes(**nodes*)
Add nodes and all connected units to the model Check that the time is the same for the model and all the units

Parameters **nodes** – EnergyNode

add_nodes_and_actors(**nodes_or_actors*)
Add nodes, actors and all connected units to the model Check that the time is the same for the model and all the units

Parameters **nodes_or_actors** – EnergyNode or Actor type

get_model_constraints_list()
Gets constraints of the model

get_model_constraints_name_list()
Gets the names of the constraints of the model

get_model_objectives_list()
Gets objectives of the model

get_model_objectives_name_list()
Gets the names of the objectives of the model

get_model_quantities_list()
Gets quantities of the model

get_model_quantities_name_list()
Gets the names of the quantities of the model

solve_and_update(*solver: pulp.apis.core.LpSolver = None*, *find_infeasible_cst_set=False*, *pareto_step=0.1*) → None
Solves the optimization model and updates all variables values.

Parameters

- **solver** (*LpSolver*) – Optimization solver
- **pareto_step** – if there are pareto objectives, you can change the step to calculate the pareto front

update_units()
Updates all units values with optimization results

omegalpes.general.optimisation.model.**check_if_unit_could_have_parent**(*unit*)
Checks if the unit has an associated parent

Parameters **unit** – unit which parents will be checked

omegalpes.general.optimisation.model.**compute_gurobi_IIS**(*gurobi_exe_path='C:\\gurobi810\\win64\\bin'*, *opt_model=None*, *MPS_model=None*)
Identifies the constraints in a .ilp file

Parameters

- **gurobi_exe_path** – Path to the gurobi solver "gurobi_cl.exe"
- **opt_model** – OptimisationModel to whom compute IIS
- **MPS_model** – name of the mps model

Then, utils methods are developped and are presented in the following module:

Input Data module

This module includes the following utils for input data management

It contains the following methods:

- select_csv_file_between_dates() : select data in a .csv file between two dates
- read_enedis_data_csv_file() : select and rearrange the data in a .csv file of Enedis (the French Distribion System Operator company), possibly between two dates

omegalpes.general.utils.input_data.**read_enedis_data_csv_file**(*file_path=None*, *start=None*, *end=None*)

Rearrange the Enedis data in cvs file in oder to have a Dataframe of the following form

DD MM YYYY HH:00 ; a DD MM YYYY HH:30 ; b ...

Parameters

- **file_path** – path of the file to rearrange
- **start** – DD MM YYYY HH:00 : first date which should be considered
- **end** – DD MM YYYY HH:00 : last date which should be considered

Returns df_list: the data as a list

omegalpes.general.utils.input_data.**resample_data**(*input_list=None*, *dt_origin=1.0*, *dt_final=1.0*, *fill_config='ffill'*, *pick_config='mean'*)

Changing data set in a dt_origin time step into data set in a dt_final time step :param input_list: list to be resample (list or dict) :param dt_origin: the time step of the input dataset (in hours) :param dt_final: the wanted time step for the output dataset (in hours) :param fill_config: choose the configuration of filling when dt_origin > dt_final by default: ffil, keeping the same data over the time steps other way: interpolate, taking into account the time steps) :param pick_config: choose the configuration of picking when dt_origin < dt_final by default: "mean" which determines the mean value of the time steps to be reduced :return: output_list: resampled list (pandas Series)

omegalpes.general.utils.input_data.**select_csv_file_between_dates**(*file_path=None*, *start='DD/MM/YYYY HH:MM'*, *end='DD/MM/YYYY HH:MM'*, *v_cols=[]*, *sep=';'*)

Select data in a .csv file between two dates

Parameters

- **file_path** – path of the file to rearrange
- **start** – DD MM YYYY HH:00 : first date which should be considered
- **end** – DD MM YYYY HH:00 : last date which should be considered
- **v_cols** – columns which should be considered

Returns df: a dataframe considering the dates

Output Data module

This module includes the following utils for output data management

It contains the following methods:

- save_energy_flows() : Save the optimisation results in a .csv file

omegalpes.general.utils.output_data.**save_energy_flows**(**nodes*, *file_name=None*, *sep='\t'*, *decimal_sep='.'*)

Save the optimisation results in a .csv file

Parameters

- **nodes** – one or several nodes from which should be collected the data
- **file_name** – name of the file to save the data
- **sep** – separator for the data
- **decimal_sep** – separator dor the decimals of the data

Plots module

This module includes the following display utils:

- plot_node_energetic_flows() : enables one to plot the energy flows through an EnergyNode
- plot_energy_mix() : enables one to plot the energy flows connected to a node
- plot_pareto2D() : enables one to plot a pareto front based on two quantities
- plot_quantity() : enables one to plot easily a Quantity
- plot_quantity_bar() : enables one to plot easily a Quantity as a bar
- sum_quantities_in_quantity() : enables one to to plot several quantities in one once the optimisation is done

omegalpes.general.utils.plots.**plot_energy_mix**(*node*)

omegalpes.general.utils.plots.**plot_node_energetic_flows**(*node*)

Description

This function allows to plot the energy flows through an EnergyNode

The display is realized :

- with histograms for production and storage flow
- with dashed curve for consumption flow

Parameters **node** – EnergyNode

omegalpes.general.utils.plots.**plot_pareto2D**(*model*, *quantity_1*, *quantity_2*, *title=None*, *legend_on=True*)

Description

Plot a Pareto front for two quantities. Before using it, you should have added in your model two objectives with the pareto parameter activated (pareto=True)

Parameters

Parameters

- **model** –
- **quantity_1** – the first quantity for the pareto front
- **quantity_2** – the second quantity for the pareto front
- **title** –

omegalpes.general.utils.plots.**plot_quantity**(*time*, *quantity*, *fig=None*, *ax=None*, *color=None*, *label=None*, *title=None*)

Description

Function that plots a OMEGAlpes.general.optimisation.elements.Quantity

Parameters

- time: TimeUnit for the studied horizon as defined in general.time
- quantity: OMEGAlpes.general.optimisation.elements.Quantity
- fig: Figure as defined in matplotlib.pyplot.Figure
- ax: axes as defined in matplotlib.pyplot.Axes
- color: color of the plot
- label: label for the quantity
- title: title of the plot

Returns

- arg1 the matplotlib.pyplot.Figure handle object
- arg2 the matplotlib.pyplot.Axes handle object
- arg3 the matplotlib.pyplot.Line2D handle object

omegalpes.general.utils.plots.**plot_quantity_bar**(*time*, *quantity*, *fig=None*, *ax=None*, *color=None*, *label=None*, *title=None*)

Description

Function that plots a OMEGALPES.general.optimisation.elements.Quantity as a bar

Attributes

- quantity is the OMEGALPES.general.optimisation.elements.Quantity
- fig could be None, a matplotlib.pyplot.Figure or Axes for multiple plots

Returns

- arg1 the matplotlib.pyplot.Figure handle object
- arg2 the matplotlib.pyplot.Axes handle object
- arg3 the matplotlib.pyplot.Line2D handle object

omegalpes.general.utils.plots.**sum_quantities_in_quantity**(*quantities_list=[]*, *tot_quantity_name='sum_quantity'*)

Description

Function that creates a new quantity gathering several values of quantities Should be used in order to plot several quantities in one once the optimisation is done

Attributes

- quantities : a list of Quantities (OMEGALPES.general.optimisation.elements.Quantity)
- tot_quantity_name : string : name of the new quantity

Returns

- tot_quantity : the new quantity created and filled

Maths module

This module includes the following maths utils

It contains the following methods:

- def_abs_value() : Define easily absolute value for quantity

omegalpes.general.utils.maths.**def_abs_value**(*quantity*, *q_min*, *q_max*)

Parameters

- **quantity** – Quantity whose absolute value is wanted
- **q_min** – Minimal value of the quantity (negative value)
- **q_max** – Maximal value of the quantity (positive value)

Returns A new Quantity whose values equal absolute values of the initial Quantity

2.4 OMEGAlpes Graphical Representation

The energy systems developed in OMEGAlpes are described following a specified graphical representation.

2.4.1 Representing an energy system

In order to link the energy units, energy nodes and arrows should be used. It is possible to highlight the variable which may be optimised by the system or which is interesting for the user. Finally, constraints and objectives can be added on the model with the following representation

Element	Symbol
Energy Unit	Name
Objective	O: obj
Constraint	C: Constraint
Power flow	
Energy node	
Considered variable	

Fig. 4: *Figure: Energy Nodes, Flows and Variable of interest representation*

2.4.2 Energy Unit representation

The energy units are described as rectangles as presented just below.

A colour may be added to specify the carrier of the energy unit like gas (yellow), heat (red), cold (blue), electricity (purple).

The symbol is adapted according to the energy unit type.

Conversion units use the former representation inside a green box as multi-carrier energy units. The representation for ElectricalToHeat, HeatPump and ReversibleConversionUnit are displayed.

Reversible units are represented as single units for simplicity purpose, even if they do include a production unit and a consumption unit like conversion units. The power flows of the production and consumption units in the reversible

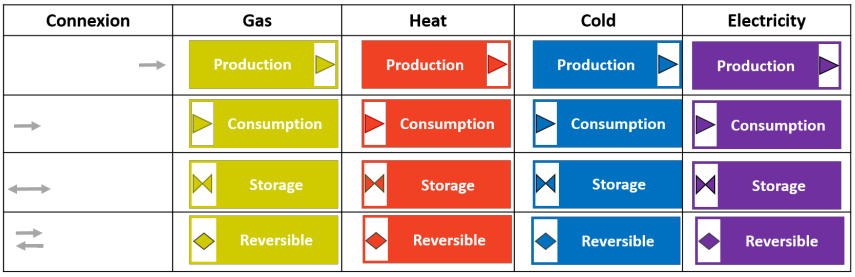

Fig. 5: *Figure: Connexion and specified carrier energy unit representation*

Unit name in OMEGAlpes	Symbol
EnergyUnit	
FixedEnergyUnit	
VariableEnergyUnit	
SquareEnergyUnit	
ShiftableEnergyUnit	
TriangleEnergyUnit	
SawTothEnergyUnit	
SeveralEnergyUnit	*nb ∈ IN
	*nb ∈ IR

Fig. 6: *Figure: Energy unit representation*

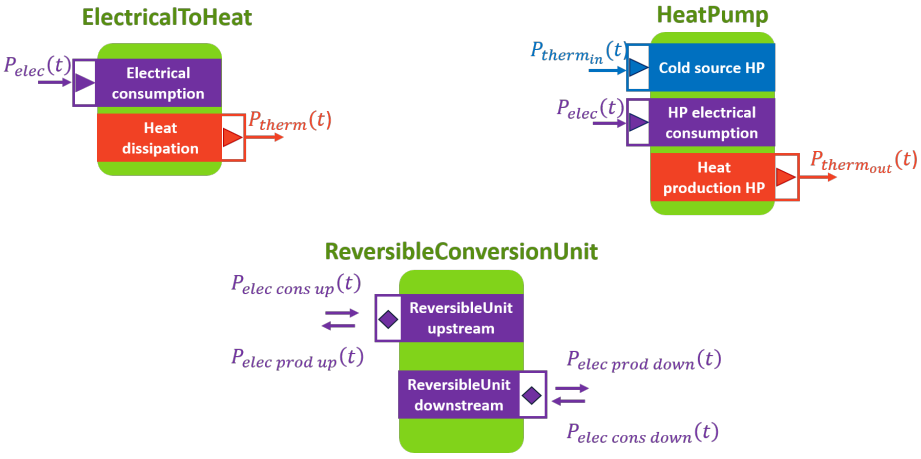

Fig. 7: *Figure: Reversible, storage and reversible conversion units representation*

unit are represented with double arrows, while storage units' power flow is represented with a single arrow with a double head. Finally, reversible conversion units are represented based on the former graphical representations.

Please, have a look to the examples to see if applications of this graphical representation OMEGAlpes Examples Documentation

---

Note: This graph representation is not used yet as a graphical user interface but we hope that it will be in the near future.

2.5 How to use OMEGAlpes

This page is under development and wil be updated

OMEGAlpes use principles will be detailed here.

In the meantime, empty templates for creating OMEGAlpes models including actors or not are available in this OMEGAlpes examples folder: https://gricad-gitlab.univ-grenoble-alpes.fr/omegalpes/omegalpes_examples/-/tree/master/case_study_template

A tutorial with linked notebook are also available.

The **'notebook folder'_** also enable to discover OMEGAlpes functionnalities, and especially this notebook about waste heat recovery.

2.6 Graphical interface

This page is under development and wil be updated

A graphical interface of OMEGAlpes has been developed in order to be able to build its model. The graphical interface currently does not include latest fucntions such as reversible energy units, and only enable to create an OMEGAlpes script, further developments are underway.

The graphical interface of OMEGAlpes is available at this adress: https://mhi-srv.g2elab.grenoble-inp.fr/OMEGAlpes-web-front-end/

The documentation linked to this interface is available in French in the gitlab wiki of OMEGAlpes-web_back-end: https://gricad-gitlab.univ-grenoble-alpes.fr/omegalpes/omegalpes-web-back-end/-/wikis/home

and OMEGAlpes-web-front-end: https://gricad-gitlab.univ-grenoble-alpes.fr/omegalpes/omegalpes-web-front-end/-/wikis/home

2.7 What's new in the latest versions

The new version of OMEGAlpes v0.4.4 is available! The release is from 20th of January 2023.

2.7.1 What's new in version 0.4.4

Version available since January 20, 2023

Bug fixed

Model

-Adding an example of modelling of an electric vehicle offering flexibility when connected to the system (with a corresponding non-regression test).

-Minimal values (1e-5) were replaced to zero, to lighten the problem formulation (non-necessary binaries are now multiplied by zero so they don't appear in the .lp file), all non-regression tests passed (with minor modifications : non-exact values due to previous numerical approximations were corrected from 1e-5 to zero).

Documentation

- Updating the docs

Contributors

Etienne Cuisinier

2.7.2 Information about the latest versions

What's new in version 0.4.3

Version available since April 29, 2022

Bug fixed

Model

Changed indexes into indices due to PuLP update causing errors.

Documentation

- Updating the authors in the README, docs and authors, and adding partners in the development of OMEGAlpes.

Contributors

Sacha Hodencq, Mathieu Brugeron,

What's new in version 0.4.2

Version available since December 3, 2021

Bug fixed

Energy

Units

New functionalities

Energy

Units

Conversion Units

- New SingleConversionUnit that can have any energy vector as input or output (but single input and single output)

Energy Nodes

General

Model

Utils

Actors

Deprecated

ElectricalToThermalConversionUnit now deprecated Energy ++++++

Actors

Documentation

- Changing the jpeg general class diagram into png to show it in the doc
- Updating the authors
- Providing the publications

Contributors

Sacha Hodencq, Mathieu Brugeron

What's new in version 0.4.1

Version available since September 21, 2021

Bug fixed

Energy

Units

- time.DT removed from the e_max and e_min calculation, since it is already taken into account in the e_tot calculation (see Issue #66)

New functionalities

Energy

Units

Storage Units

- Adding pd_max and pc_max ratio to link it to the storage capacity when optimising it with the minimize_capacity objective. Adding pc_max and pd_max quantities
- Modifying the min_capacity objective by taking into account a maximum capacity and a initial value to SOC
- Adding a minimize_gwp objective

Energy Nodes

General

Model

- Modifying the Pareto creation method by assigning a time limite of 60 seconds for the solving of each optimization problem

Utils

Actors

Producer Actor

Prosumer Actor

- adding the operated_storage_unit_list in order to allow the consideration of storage units into the prosumer area of responsibility
- add a min_capacity actor objective which activate the min_capacity for all storage units in the prosumer area of responsability

HeatGridOperator

- allowing the StorageUnit class into the operated_unit_type_tuple parameter
- add a min_capacity actor objective which activate the min_capacity for all storage units in the prosumer area of responsability

ElectricityGridOperator

- allowing the StorageUnit class into the operated_unit_type_tuple parameter
- add a min_capacity actor objective which activate the min_capacity for all storage units in the prosumer area of responsability

Deprecated

Energy

Consumption Units

Production Units

Actors

Documentation

Completing OMEGAlpes documentation with exergy module and actors package description. Adding pages about the use of OMEGAlpes and the graphical interface

Contributors

Sacha Hodencq, Mathieu Brugeron, Lou Morriet

What's new in version 0.4.0

Version available since 05 01 2021

Bug fixed

- PuLP library version changed from >=1.6.10 to 2.1: changed imports in *general.optimisation.model*, as well as requirements.txt, setup.py and associated docs.

New functionalities

Energy

Units

- e_max and e_min parameters now add technical constraints (set_e_min and set_e_max) when set to a value.
- e_tot upper bound is now set as the p_max value times the number of hours over the studied time period. The lower bound is either set to 0, or minus the upper bound if the unit is a storage unit.
- the *add* methods put in place with parameters are now *_add* methods.
- set_e_max constraint changed to set_e_max_period (respectively e_min)

Consumption Units

- add imaginary parameter in SeveralConsumptionUnit

Production Units

- add new parameters : particle_emission (quantity) and rr_energy (True/False renewable and recovery energy unit)
- add imaginary parameter in SeveralProductionUnit

Energy Nodes

- add get_input_poles and get_output_poles

General

Model

- add lpfics (Linear Programming : Find Incompatible Constraint Sets) to

help finding incompatible constraint sets in OMEGAlpes projects

Utils

- add plot_2D_pareto method

Actors

Update the Actor structure - move Prosumer class into prosumer_actors.py - move Supplier class into supplier_actors.py

Project Developer Actors

- create a ProjectDeveloper class as a new type of actor

Operator Actor

- add grid_operator_actors.py
- add prosumer_actors.py
- add supplier_actors.py

Consumer Actor

- use add_power_consumption_total_minimum or add_power_consumption_by_unit_minimum instead of add_power_consumption_minimum
- use add_power_consumption_total_maximum or add_power_consumption_by_unit_maximum instead of add_power_consumption_maximum

Producer Actor

- use add_power_production_total_minimum or add_power_production_by_unit_minimum instead of add_power_production_minimum
- use add_power_production_total_maximum or add_power_production_by_unit_maximum instead of add_power_production_maximum
- add add_temporary_stop

Deprecated

- the name of the function set_operating_time_range changed to add_operating_time_range for code consistency. set_operating_time_range is now deprecated.

Energy

Consumption Units

- delete SeveralImaginaryConsumptionUnit (use imaginary parameter in SeveralConsumptionUnit instead)

Production Units

- delete SeveralImaginaryProductionUnit (use imaginary parameter in SeveralProductionUnit instead)

Actors

Consumer Actor

- add_power_consumption_minimum (replaced by add_power_consumption_total_minimum)
- add_power_consumption_maximum (replaced by add_power_consumption_total_maximum)

Producer Actor

- add_power_production_minimum (replaced by add_power_production_total_minimum)
- add_power_production_maximum (replaced by add_power_production_total_minimum)

Contributors

Lou Morriet, Sacha Hodencq

What's new in version 0.3.1

Version available since 13th March 2020

Bug fixed

- Change remaining pmin and pmax parameters into p_min and p_max of EnergyUnits in conversion units
- Add verbose parameter set to False in general/utils/input_data/resample_data() Timeunit
- Change expected type of *units in connect_units() from list to EnergyUnit

New functionalities

General

Time

- Add verbose parameter added to TimeUnit in order to be able not to print the studied time period.

Elements

- Add three classes to make a distinction between different kind of constraints: DefinitionConstraint, TechnicalConstraints and ActorConstraint. Considering the DynamicConstraints, three other classes have been created:
  - DefinitionDynamicConstraint,
  - TechnicalDynamicConstraints, and
  - ActorDynamicConstraint.

Deprecated

Class

- Deprecate ExternalConstraint and ExtDynConstraint in module Elements

Contributors

Lou Morriet, Sacha Hodencq

What's new in version 0.3.0

Version available since 18th February 2020

Bug fixed

- Change pmin and pmax parameters of VariableEnergyUnits and its derivatives (SeveralEnergyUnit, VariableConsumptionUnit, SeveralConsumptionUnit, VariableProductionUnit, SeveralProductionUnit) into p_min and p_max

New Functionalities

Energy

Units

- EnergyUnit has "minimize_exergy_destruction" and "minimize_exergy" objectives
- FixedEnergyUnit takes into account Dataframe for power values
- ElectricalConversionUnit created in conversion_units
- ReversibleUnit created in a dedicated package in the energy.units folder. ReversibleUnit is a new type of energy unit that can both produce or consume energy, but not at the same time.
- AssemblyUnits created in the energy_units package as an assembly of one or several production, consumption and/or reversible unit. It is the parent class of both conversion units and reversible units.
- AssemblyUnit has "minimize_exergy_destruction" objective
- ReversibleConversionUnit created in conversion units, enabling reversible power flows (i.e. an electrical transformer). It is an AssemblyUnit of two reversible units, one qualified as *upward* and another *downward*.
- StorageUnit has a new attribute self_disch_t

Exergy

- Exergy module created. Integration of the exergy approach in OMEGAlpes. With the code of this branch, OMEGAlpes will be able to quantify thermal and electrical exergy, as well as the destroyed exergy within any production, consumption, storage or conversion unit.

General

Optimisation

- HourlyDynamicConstraint deprecated and changed into DailyDynamicConstraint, now used in set_operating_hours() (that replaced add_operating_hours())

Utils

- Titles as parameters in plot_quantities()
- Add function get_value() to get int or list
- Add get_value_with_dates() to get a dataframe
- Add resample_data() to change the time step for a data set

Deprecated

Methods

- Deprecate add_operating_time_range() in class EnergyUnit

Class

- Deprecate HourlyDynamicConstraint in module Elements

Contributors

Lou Morriet, Sacha Hodencq, Mathieu Brugeron, Jaume Fito

2.8 OMEGAlpes Examples

Please click on the following link to have a look to OMEGAlpes examples and study cases: OMEGAlpes Examples Documentation

CHAPTER 3

Licence

OMEGAlpes is open-source and written in Python with the licence Apache 2.0. This documentation is licensed with the Creative Commons Attribution 4.0 International (CC BY 4.0)

CHAPTER 4

Authors:

See AUTHORS in OMEGAlpes repository.

CHAPTER 5

Acknowledgments

Vincent Reinbold - Library For Linear Modeling of Energetic Systems : https://github.com/ReinboldV

This work has been partially supported by the CDP Eco-SESA receiving fund from the French National Research Agency in the framework of the “Investissements d’avenir” program (ANR-15-IDEX-02) and the VALOCAL project (CNRS Interdisciplinary Mission and INSIS)

Python Module Index

o

Index

A

C

D

E

F

G

H

I

L

M

O

P

Q

R

S

T

U

V

W

Z