tamos

BNerot

Nov 09, 2022

CONTENTS

CHAPTER

ONE

GATHERING COMPONENTS

tamos.Hub([name, components, ...])	
tamos.TimeSettings(n[, step_value, ...])	
tamos.MILPModel(hubs, time_settings[, name])	

1.1 tamos.Hub

class tamos.**Hub**(*name=None*, *components=None*, *components_assemblies=None*, *interface_masks=None*)

__init__(*name=None*, *components=None*, *components_assemblies=None*, *interface_masks=None*)

Gathers components and allow power exchanges through a dedicated interface.

Enables simple definition of MILP constraints using the *components_assemblies* and *interface_masks* attributes.

Parameters

- **name** (*str, optional*) –
- **components** (*list of storage, production or element_IO instances, optional*) –
- **components_assemblies** (*list of 3-tuple objects (n_min, n_max, components), optional*) – Each 3-tuple is such that:
  - *n_min* (*n_max*) is the minimum (maximum) number of components from *components* that must be installed in the hub.
  - ***components* is a component or list of production, storage or element_IO components.**
    If one component of *components* is not one of hub components, the 3-tuple (n_min, n_max, components) is ignored during constraints declaration.
- **interface_masks** (*list of InterfaceMask instances, optional*) – Instances in *interface_masks* that do not refer to a component of the hub will be ignored during constraints declaration.

Examples

```
>>> tms.Hub(components=[heat_load_1, heat_load_2, heat_load_3, electric_heater_
↪1, electric_heater_2, electricity_grid],
...         components_assemblies=[(1, 2, [heat_load_1, heat_load_2]), (1, 1,␣
↪heat_load_3), (0, 1, [electric_heater_1, electric_heater_2])])
```

At least one of the first and second loads must be used. The third load must be used. At most one of the two heaters must be used.

```
>>> tms.Hub(components=[heat_load, electric_heater, electricity_grid, heat_
↪grid],
...         components_assemblies=[(0, 1, [electric_heater, heat_grid]), (1, 1,␣
↪heat_load)])
```

The load must be used. Either electric_heater or heat_grid can be used.

Methods

__init__([name, components, ...])	Gathers components and allow power exchanges through a dedicated interface.
change_components(components[, add, remove])	Change the components that can be installed in the hub.
change_interface_masks(interface_masks[, ...])	Change the InterfaceMasks instances related to the hub.
describe()	Show an exhaustive description of the hub.

Attributes

components	production, storage and element_IO components in the hub.
components_assemblies	Components assemblies of the hub.
element_IOs	element_IO components in the hub.
interface_masks	InterfaceMask instances of the hub.
name	str.
possibly_connected_networks	network components that include the hub with at least one edge with a connection status is different from 'network.no_connection'.
productions	production components in the hub.
storages	storage components in the hub.

change_components(*components*, *add=False*, *remove=False*)

Change the components that can be installed in the hub.

Parameters

- **components** (*instance or list of instances of production, storage and element_IO components.*) –
- **add** (*bool, optional*) – If True, *components* are added to already defined components in hub.

- **remove** (*bool, optional*) – If True, all instances from *components* that are components of the hub are removed from the hub.

Notes

If *add* and *remove* are False, hub components are replaced by *components*. *add* and *remove* cannot be both True.

change_interface_masks(*interface_masks*, *add=False*, *remove=False*)

Change the InterfaceMasks instances related to the hub.

Parameters

- **interface_masks** (*instance or list of instances of InterfaceMask.*) – Instances in *interface_masks* that do not refer to a component of the hub will be ignored during constraints declaration.
- **add** (*bool, optional*) – If True, *components* are added to already defined components in hub.
- **remove** (*bool, optional*) – If True, all instances from *components* that are components of the hub are removed from the hub.

Notes

If *add* and *remove* are False, hub components are replaced by *components*. *add* and *remove* cannot be both True.

property components

production, storage and element_IO components in the hub.

These can be modified using the *hub.change_components* method.

property components_assemblies

Components assemblies of the hub.

Must be provided as a list of 3-tuple objects (n_min, n_max, components) where:

- *n_min* (*n_max*) is the minimum (maximum) number of components from *components* that must be installed in the hub.
- *components* is a component or list of production, storage or element_IO components. If one component of *components* is not one of hub components, the 3-tuple (n_min, n_max, components) is ignored during constraints declaration.

Examples

```
>>> hub.components_assemblies = [(1, 2, [heat_load_1, heat_load_2]), (1, 1,␣
↪heat_load_3), (0, 1, [electric_heater_1, electric_heater_2])])
```

At least one of the first and second loads must be used. The third load must be used. At most one of the two heaters must be used.

```
>>> hub.components_assemblies = [(0, 1, [electric_heater, heat_grid]), (1, 1,␣
↪heat_load)])
```

The load must be used. Either electric_heater or heat_grid can be used.

describe()

Show an exhaustive description of the hub.

property element_IOs

element_IO components in the hub. These can be modified using the *hub.change_components* method.

property interface_masks

InterfaceMask instances of the hub.

These can be modified using the *hub.change_interface_masks* method.

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property possibly_connected_networks

network components that include the hub with at least one edge with a connection status is different from 'network.no_connection'.

property productions

production components in the hub.

These can be modified using the *hub.change_components* method.

property storages

storage components in the hub.

These can be modified using the *hub.change_components* method.

1.2 tamos.TimeSettings

class tamos.**TimeSettings**(*n*, *step_value=1*, *system_lifetime=40*)

__init__(*n*, *step_value=1*, *system_lifetime=40*)

Defines temporal parameters used for the optimization of energy systems. Provide helper function to implement a reduced-complexity temporal approach regarding system operation. See examples section for in-depth understanding.

Parameters

- **n** (*int*) – The length of the operation period.
- **step_value** (*int, optional, default 1*) – Length of each time step of the operation period, in hours. Required to be consistent with some component physical properties.
- **system_lifetime** (*int, optional, default 40*) – Number of periods considered for economic amortization of components. Is related to the property "Discount rate (%)" of components. See method *component.compute_actualized_cost*. The default value *40* is a typical value relevant for an annual operation, i.e. such that *n* * *step_value* = 8760

Examples

```
>>> time_settings = TimeSettings(n=8760, step_value=1, system_lifetime=30)
```

The operation period has a length of n = 8760. Each time serie parameter of components must be of length n. Their values last all 1 hour (*step_value* = 1). 8760 values of 1 hour is one year. Economic amortization is calculated for 30 periods, i.e. 30 years. Up to now, *time_settings* does not define any relevant time_step for operation. They must be added using the methods:

- add_regular
- add_extreme_values
- add_large_diff

```
>>> time_settings.add_regular(5)
```

Every index multiple of 5 is selected for operation. At this point, the three first time steps (for instance) are defined by the following sets of indexes:

- {0, 1, 2, 3, 4}
- {5, 6, 7, 8, 9}
- {10, 11, 12, 13, 14}
- ...

This leads to consider approximately 8760/5 = 1752 time steps for operation. Decision variables are indexed on the first element of these index sets. Each parameter of the model given as a numpy.ndarray instance of length 8760 is averaged according to these time steps, defining a new array *new_array* of length 1752. For instance:

- new_array[0] = array[0:5].mean()
- new_array[1] = array[5:10].mean()
- new_array[2] = array[10:15].mean()
- ...

```
>>> time_settings.add_extreme_values(array=load_values, number_max=3)
```

The indexes of the 3 largest values of *load_values* are added to the operation temporal vector, splitting existing time steps. For instance, if 2 is one of these 3 indexes, the sets of indexes defining time steps become:

- {0, 1}
- {2}
- {3, 4}
- {5, 6, 7, 8, 9}
- {10, 11, 12, 13, 14}
- ...

Now, any array will be average according to:

- new_array[0] = array[0:2].mean()
- new_array[1] = array[2]

- new_array[2] = array[3:5].mean()
- new_array[3] = array[5:10].mean()
- ...

```
>>> time_settings.add_extreme_values(array=temperature_values, number_min=2,␣
↪number_max=3)
```

The indexes of the two smallest and three largest values of *temperature_values* are added to the operation temporal vector.

Methods

__init__(n[, step_value, system_lifetime])	Defines temporal parameters used for the optimization of energy systems.
add(indexes)	Adds specific indexes to the operation temporal vector.
add_extreme_values(array[, number_min, ...])	Add the time steps corresponding to minimum and maximum values of an array to the operation temporal vector.
add_large_diff(array, number_max)	Apply 'add_extreme_values' to the absolute value of the temporal derivative of 'array'.
add_regular(step)	Add every multiple of *step* to the operation temporal vector.
plot_array_aggregation(array)	Plot the aggregated version of an array, aggregation being performed according to the *time_steps* attribute.
prepare_array(array[, raise_error])	Checks that array length complies with the length of the operation period.

Attributes

time_steps	Presents each time step of the temporal operation vector defined by its set of indexes.

add(*indexes*)

Adds specific indexes to the operation temporal vector.

Parameters

indexes (*int or list of int*) – All values greater than *n* will be ignored. Each value of *indexes* defines a new time step, i.e. time series parameters of components are not averaged on these indexes.

add_extreme_values(*array*, *number_min=None*, *number_max=None*)

Add the time steps corresponding to minimum and maximum values of an array to the operation temporal vector.

Parameters

- **array** (*numpy.ndarray*) –
- **number_min** (*int, optional*) – The indexes of the *number_min* (*numer_max*) smallest (largest) values of *array* will be added to the operation temporal vector.

- **number_max** (*int, optional*) – The indexes of the *number_min* (*numer_max*) smallest (largest) values of *array* will be added to the operation temporal vector.

add_large_diff(*array*, *number_max*)

Apply 'add_extreme_values' to the absolute value of the temporal derivative of 'array'.

Parameters

- **array** (*numpy.ndarray*) –
- **number_max** (*int*) – The indexes of the *number_max* largest values to be added to the operation temporal vector.

Notes

Temporal derivative of *array* is d_array defined as:
d_array(t) = abs(array(t+1)-array(t))

add_regular(*step*)

Add every multiple of *step* to the operation temporal vector.

Parameters
step (*int*) –

plot_array_aggregation(*array*)

Plot the aggregated version of an array, aggregation being performed according to the *time_steps* attribute.

This aggregation is how arrays are dealt with in *MILPModel* instances.

Parameters
array (*numpy.ndarray*) –

prepare_array(*array*, *raise_error=False*)

Checks that array length complies with the length of the operation period. If array is too long, this method acts depending on *raise_error*. If array is too short, an attributeError is raised.

Parameters

- **array** (*numpy.ndarray*) – The array to format.
- **raise_error** (*bool, optional, default False*) – Describes how *array* is processed when its length exceeds the length of the operation period:
  - If True, an AttributeError is raised.
  - If False, *array* is sliced so that its first n elements are returned.

Returns

- * AttributeError if *raise_error* is True and array is too long, or array is too short.
- * The first n values of *array* otherwise.

property time_steps

Presents each time step of the temporal operation vector defined by its set of indexes.

1.3 tamos.MILPModel

class tamos.**MILPModel**(*hubs:* Hub, *time_settings*, *name=None*)

__init__(*hubs:* Hub, *time_settings*, *name=None*)

Manages the declaration and solving processes of the MILP problem associated to the components of hubs *hubs*.

Parameters

- **hubs** (*Hub or list of Hub*) – production, storage, element_IO and network components related to *hubs* are sized and operated using *time_settings* parameters.
- **time_settings** (*TimeSettings instance*) –
- **name** (*str, optional*) –

Methods

__init__(hubs, time_settings[, name])	Manages the declaration and solving processes of the MILP problem associated to the components of hubs *hubs*.
declare_constraints_and_KPIs()	Declare constraints and KPIs associated with components related to *hubs*.
declare_max_KPI_constraint(kind, max_value)	Constrains one of the objective KPI.
declare_variables()	Declare all the decision variables associated with components related to *hubs*.
remove_constraints_and_KPIs([component])	[experimental, unstable] Removes from the MILP model some constraints and KPIs.
remove_max_KPI_constraint(kind)	Removes the limit on the objective KPI of kind *kind*, if it exists.
solve(kind[, MIP_gap, threads, timelimit])	Solves the MILP model minimizing the objective function of kind *kind*.

Attributes

components_assemblies	Components assemblies of the model.
description	Stores a description of the model.
hubs	production, storage, element_IO and network components related to *hubs* are sized and operated using *time_settings* parameters.
name	Name of this MILPModel instance.

property components_assemblies

Components assemblies of the model.

Must be provided as a list of 3-tuple objects (n_min, n_max, components) where:

- *n_min* (*n_max*) is the minimum (maximum) number of components from *components* that must be installed, all hubs of *hubs* included.

- *components* is a component or list of production, storage or element_IO components. For any component of *components*, if this component is not in at least one hub of this MILPModel instance, the 3-tuple (n_min, n_max, components) is ignored during constraints declaration.

Examples

```
>>> hub_1 = Hub(components=[heat_load_1, heat_load_2])
>>> hub_2 = Hub(components=[heat_load_1])
>>> hub_3 = Hub(components=[heat_load_2])
>>> MILPModel = MILPModel(hubs=[hub_1, hub_2, hub_3])
>>> MILPModel.components_assemblies = [(0, 1, heat_load_1), (2, 3, [heat_load_
↪1, heat_load_2])]
```

heat_load_1 might be used at most one time, all hubs [hub_1, hub_2, hub_3] included. the number of times heat_load_1 or heat_load_2 are used in all hubs [hub_1, hub_2, hub_3] is greater than 2 but smaller than 3.

declare_constraints_and_KPIs()

Declare constraints and KPIs associated with components related to *hubs*. All KPIs are defined, i.e. covering each kind of objective function (Eco, Exergy, CO2).

declare_max_KPI_constraint(*kind*, *max_value*)

Constrains one of the objective KPI. The values of these objective KPI are described in the *solution_summary* attribute of ResultsExport instances. This constraint can be removed by a call to *remove_max_KPI_constraint(kind)*.

Parameters

- **kind** (*{'Eco', 'CO2', 'Exergy'}*) –
- **max_value** (*float*) –

Notes

1. This method calls *remove_max_KPI_constraint(kind)* if necessary to remove any already defined constraint regarding kind *kind*.
2. A constraint on kind *kind* can be defined no matter the kind of objective function.
3. More than one kind can be constrained at a time.

Examples

```
>>> MILPModel.declare_max_KPI_constraint("Eco", 1e6)
```

The total cost of the system over its lifetime cannot exceed 1 million euros. >>> MILPModel.declare_max_KPI_constraint("CO2", 2e5)

The net CO2 emissions of the system over the operation period (net = entering - exiting) cannot exceed 200 tEqCO2 >>> MILPModel.declare_max_KPI_constraint("Exergy", 1e5)

The net exergetical potential consumed by the system over the operation period (net = entering - exiting) cannot exceed 1e5 kWh. >>> MILPModel.declare_max_KPI_constraint("Eco", -1e5)

The system must be economically profitable and generate at least 100 000 euros over its lifetime.

`declare_variables()`

Declare all the decision variables associated with components related to *hubs*.

`property description`

Stores a description of the model. These properties are used to gather ResultsExport objects written on disk and create ResultsBatch objects. They define the *relevant_descriptors* attribute of ResultsBatch objects.

Some of these properties are automatically declared by tamos. The other are user-defined. description: dict

- keys are str
- values are int, float or str

`property hubs`

production, storage, element_IO and network components related to *hubs* are sized and operated using *time_settings* parameters.

`property name`

Name of this MILPModel instance. If None, name is the current time.

`remove_constraints_and_KPIs(component='all')`

[experimental, unstable] Removes from the MILP model some constraints and KPIs. The next call to *declare_constraints_and_KPIs* will only redefine the missing constraints and KPIs, saving model declaration time. Relevant after a component modification.

Parameters

component (*Four possible kinds of arg are accepted:*) –

- 'all': all constraints and KPIs of the MILP model are removed (default)
- None: assembly_constraints of this MILPModel instance are removed. To remove and define constraints on Eco, Exergy and CO2 KPIs, please use the dedicated methods.
- production, storage, element_IO, or network component: constraints and KPIs of the corresponding component are removed, no matter the hub(s) they belong or are related to.
- hub: constraints and KPIs related to the hub are removed. This includes:
  - hub assemblies constraints
  - constraints and KPIs of all production, storage and element_IO components of hub
  - hub interface constraints (the ones that bind every element exchanges between components)

Notes

1. In case component is a production, storage, element_IO, or network component. Constraints and KPIs removal of component does not affect the contribution of the component to the hub interface. Unconstrained energy flows will occur if constraints and KPIs are not declared again after call to this method.
2. Some model modifications require a complete model redeclaration (variables, constraints, KPIs). These cases are:
   - Some components were added or removed from hubs (production, storage, element_IO)
   - New networks are used
   - *time_steps* attribute of *time_settings* was modified

In these cases, variable redeclaration is automatically performed.

3. Regarding constraints and KPIs, *Cost* instances are bound to the element_IO component they describe.
4. Regarding constraints and KPIs, *InterfaceMask* instances are bound to the component they describe.

remove_max_KPI_constraint(*kind*)

Removes the limit on the objective KPI of kind *kind*, if it exists.

Parameters
kind (*{'Eco', 'CO2', 'Exergy'}*) –

solve(*kind*, *MIP_gap=0.0001*, *threads=0*, *timelimit=43200*)

Solves the MILP model minimizing the objective function of kind *kind*. Model is solved using the Cplex solver. To use another MILP solver, please export the model in LP or MPS format using a ResultsExport instance.

Parameters

- **kind** (*{'Eco', 'CO2', 'Exergy'}*) –
- **MIP_gap** (*float, optional, default 1e-4*) – 0 <= MIG_gap <= 1 According to Cplex documentation, gap between the best integer objective and the objective of the best node remaining.
- **threads** (*int, optional, default 0*) – 0 <= threads According to Cplex documentation, maximal number of parallel threads that will be invoked by any CPLEX parallel optimizer.
  - If 0, let Cplex decide
  - Else, uses up to *threads* threads
- **timelimit** (*int, optional, default 43200*) – 0 < timelimit According to Cplex documentation, maximum time, in seconds, for a call to an optimizer.

Notes

Other Cplex parameters can be set by accessing the attribute *_model_data.mdl.parameters* of this *MILPModel* instance. For instance: >>> MILPModel._model_data.mdl.parameters.mip.cuts.flowcovers.set(2)

CHAPTER

TWO

ENERGY VECTORS

2.1 Elements

tamos.element.ElectricityVector([name])	
tamos.element.FuelVector(exergy_factor[, name])	
tamos.element.ThermalVector(temperature[, name])	

2.1.1 tamos.element.ElectricityVector

class tamos.element.**ElectricityVector**(*name=None*)

__init__(*name=None*)

ElectricityVector instances are only defined by an exergy factor (default to 1). Power is exchanged in kW.

Parameters
name (*str, optional*) –

Notes

The exergy factor can be redefined using the *exergy_factor* attribute.

Methods

__init__([name])	ElectricityVector instances are only defined by an exergy factor (default to 1).
get_vectors()	Returns this instance.

Attributes

exergy_factor	Quantity of exergy associated with a 1 kW power flow of this element.
name	str.

property exergy_factor

Quantity of exergy associated with a 1 kW power flow of this element. Theoretically positive, usually smaller than 1. In kW/kW.

int, float or numpy.ndarray 0 <= exergy_factor

Notes

FuelVector instances are exchanged between components considering their LHV value.

get_vectors()

Returns this instance.

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! ” # $ % & , . ; ? @ _ ‘ ’ { } ~.

Type
Name of the instance

2.1.2 tamos.element.FuelVector

class tamos.element.**FuelVector**(*exergy_factor*, *name=None*)

__init__(*exergy_factor*, *name=None*)

A FuelVector instance may describe every energy vector whose exergy factor does not depend on a temperature (i.e. ThermalVector, ThermalVectorPair) or is not unity (i.e. ElectricityVector). Power is exchanged in kW.

These vectors are usually associated with a combustion flame temperature which can be used by the user to define the exergy factor according to: exergy_factor = 1 - dead_state_temperature / flame temperature.

Parameters

- **exergy_factor** (*float*) – 0 <= exergy_factor
- **name** (*str, optional*) –

Notes

1. The dead state temperature is the reference temperature for calculation of exergy factor of ThermalVectorPair and ThermalVector instances. It can be modified using the *set_dead_state_temperature* function of tamos.element. 2. Some typical flame temperatures are stored in the class attribute *FuelVector.typical_flame_temperature* (natural gas, biomass). 3. Power flows related to a FuelVector take for reference the lower heating value of the fuel. This impacts the efficiency models in production components. 4. The exergy factor can be redefined using the *exergy_factor* attribute.

Methods

__init__(exergy_factor[, name])	A FuelVector instance may describe every energy vector whose exergy factor does not depend on a temperature (i.e.
get_vectors()	Returns this instance.

Attributes

exergy_factor	Quantity of exergy associated with a 1 kW power flow of this element.
name	str.
typical_flame_temperature	

property exergy_factor

Quantity of exergy associated with a 1 kW power flow of this element. Theoretically positive, usually smaller than 1. In kW/kW.

int, float or numpy.ndarray 0 <= exergy_factor

Notes

FuelVector instances are exchanged between components considering their LHV value.

get_vectors()

Returns this instance.

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! ” # $ % & , . ; ? @ _ ‘ ’ { } ~.

Type
Name of the instance

2.1.3 tamos.element.ThermalVector

class tamos.element.**ThermalVector**(*temperature*, *name=None*)

__init__(*temperature*, *name=None*)

ThermalVector instances may:

- be used in a ThermalVectorPair instance.
- describe an infinite thermal reservoir. It is used to model a thermal source or sink that would not be affected by the energy taken from or given to it.

```
>>> ThermalVector(temperature=temperature_profile)
```

For example, it is relevant to model the ambiant air using a ThermalVector. Such a vector, associated with a Grid instance (i.e. an element_IO component), would describe the heat rejected by heat pumps on their condenser side.

```
>>> Grid(element=air, exergy_count=False)
```

In that case, exergy count migh not be relevant thus it can be disabled using the *exergy_count* argument of the Grid class.

Power is exchanged in kW.

Parameters

- **temperature** (*int, float or numpy.ndarray*) – In Kelvins (K). Used in:
  - the efficiency definition of some production components (e.g. heat pumps)
  - the *exergy_factor* attribute
  - power and mass balance if this instance is used in a THermalVectorPair
- **name** (*str, optional*) –

Notes

1. Two ThermalVector instances having the same temperature are still considered different.
2. The dead state temperature is the reference temperature for calculation of exergy factor of ThermalVectorPair and ThermalVector instances. It can be modified using the *set_dead_state_temperature* function of tamos.element. The exergy factor calculation is done as follow: exergy_factor = 1 - dead_state_temperature / temperature
3. The exergy factor can be user-defined using the *exergy_factor* attribute.

Methods

__init__(temperature[, name])	ThermalVector instances may:
get_vectors()	Returns this instance.

Attributes

exergy_factor	Quantity of exergy associated with a 1 kW power flow of this element.
name	str.
temperature	int, float or numpy.ndarray In Kelvins (K)

property exergy_factor

Quantity of exergy associated with a 1 kW power flow of this element. Theoretically positive, usually smaller than 1. In kW/kW.

int, float or numpy.ndarray 0 <= exergy_factor

Notes

FuelVector instances are exchanged between components considering their LHV value.

get_vectors()

Returns this instance.

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ‘ ’ { } ~.

Type
Name of the instance

property temperature

int, float or numpy.ndarray In Kelvins (K)

2.2 Utilities

tamos.element.fetch_TVP(in_TV, out_TV[, Cp, ...])	This method defines a ThermalVectorPair instance by its incoming and outcoming ThermalVector instances.
tamos.element.get_existing_TVPs()	
tamos.element.get_dead_state_temperature()	The dead state temperature is the reference temperature automatically used in calculation of attribute *exergy_factor* of ThermalVector and ThermalVectorPair instances.
tamos.element.set_dead_state_temperature(...)	Assign a new value to the dead state temperature, which is common to all elements.

2.2.1 tamos.element.fetch_TVP

tamos.element.**fetch_TVP**(*in_TV*, *out_TV*, *Cp=None*, *name=None*)

This method defines a ThermalVectorPair instance by its incoming and outcoming ThermalVector instances.

A ThermalVectorPair instance aims to represent a fluid (typically water) receiving or giving thermal energy to a subsystem by going through a sensible heat exchange. The fluid entering the subsystem is described by ThermalVector *in_TV* with temperature in_TV.temperature. The fluid exiting the subsystem is *out_TV*, with temperature out_TV.temperature. Power is exchanged in kW.

Parameters

- **in_TV** (ThermalVector) –
- **out_TV** (ThermalVector) –
- **Cp** (*int, float or numpy.ndarray, optional*) –
- **name** (*str, optional*) –

Returns

- ** If a ThermalVectorPair instance defined by (in_TV, out_TV) already exists, returns this instance –*
  - If *name* is None, current name of the instance is kept.

  - Else, the instance is renamed.
- * *Else, returns a new ThermalVectorPair instance.*

Notes

1. It is convenient to speak about ThermalVectorPair as 'TVP'.
2. A positive power flow associated with a TVP (in_TV, out_TV) in a component describes a positive mass flow rate of ThermalVector in_TV (mass enters the system) and the negated same mass flow rate of ThermalVector out_TV (mass leaves the system).
3. If (in_TV, out_TV) defines a ThermalVectorPair instance *TVP1*, the ThermalVectorPair (out_TV, in_TV) is called 'the invert of *TVP1*', for example *TVP2*, and can be accessed following two ways:

```
>>> TVP2_first = fetch_TVP(in_TV=TVP1.out_TV, out_TV=TVP1.in_TV)
>>> TVP2_second = ~TVP1
>>> TVP2_first is TVP2_second
True
```

4. Though balances are based on power flows in kW at the component scale, *Cp* is required to perform mass balances at hub interface scale.
5. Cp should depend on the temperature of *in_TV* and *out_TV*. In practice, Cp variations are small on the temperature ranges [0°C, 110°C]. e.g.
   - max: Cp(T=110°C)=4228.3 J/(kg.K)
   - min: Cp(T=40°C)=4179.6 J/(kg.K)
6. The *exergy_factor* attribute is calculated as follow: exergy_factor = 1 - dead_state_temperature * log(out_TV.temperature/in_TV.temperature) / (out_TV.temperature-in_TV.temperature)

2.2.2 tamos.element.get_existing_TVPs

tamos.element.**get_existing_TVPs**()

2.2.3 tamos.element.get_dead_state_temperature

tamos.element.**get_dead_state_temperature**()

The dead state temperature is the reference temperature automatically used in calculation of attribute *exergy_factor* of ThermalVector and ThermalVectorPair instances. It can be modified using the *set_dead_state_temperature* function of tamos.element.

Return type
The dead state temperature, in Kelvins (K).

2.2.4 tamos.element.set_dead_state_temperature

tamos.element.**set_dead_state_temperature**(*dead_state_temperature*)

Assign a new value to the dead state temperature, which is common to all elements. Already defined elements are not affected by this change.

Parameters

dead_state_temperature (*int, float or numpy.ndarray*) –

CHAPTER

THREE

PRODUCTION COMPONENTS

tamos.production.AbsHP(energy_drive, ...[, ...])	
tamos.production.BiomassBoiler(...[, ...])	
tamos.production.CHP(energy_source, ...[, ...])	
tamos.production.COPModels()	
tamos.production.CompHP(energy_drive, ...[, ...])	
tamos.production.DryCooler(energy_drive, ...)	
tamos.production.ElectricHeater(...[, ...])	
tamos.production.ElementConverter(element_1, ...)	
tamos.production.FPSolar(energy_sink, ...[, ...])	
tamos.production.GasBoiler(energy_source, ...)	
tamos.production.HeatExchanger(...[, ...])	
tamos.production.Pump(element_1, element_2, ...)	

3.1 tamos.production.AbsHP

class tamos.production.**AbsHP**(*energy_drive*, *energy_source*, *energy_sink*, *electricity_aux*, *properties*, *given_sizing=None*, *pump_consumption=0.026*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*energy_drive*, *energy_source*, *energy_sink*, *electricity_aux*, *properties*, *given_sizing=None*, *pump_consumption=0.026*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

AbsHP components describe absorption heat pumps. The COP model is an adaptation of (Boudéhenn et al., 2016)[2].

[2] Boudéhenn F, Bonnot S, Demasles H, Lefrançois F, Perier-Muzet M, Triché D. Development and Performances Overview of Ammonia-water

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is either *energy_sink* or *energy_source* depending on the *ref_production* attribute.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_drive** (*ThermalVectorPair*) – Thermal flow that is cooled down for the desorption process.
- **energy_source** (*ThermalVectorPair*, ThermalVector) – Element that gives thermal energy. Must be cooled down if ThermalVectorPair.
- **energy_sink** (*ThermalVectorPair*, ThermalVector) – Element that receives thermal energy. Must be warmed up if ThermalVectorPair.
- **electricity_aux** (ElectricityVector) – Electricity consumed by the machine.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **pump_consumption** (*float, optional, default 0.026*) – Electricity consumption of the heat pump, in proportion of the power extracted from *energy_source*.
- **name** (*str, optional*) –
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.

Absorption Chillers with Cooling Capacities from 5 to 100 kW. Energy Procedia 2016;91:707–16. https://doi.org/10.1016/j.egypro.2016.06.234.

- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.

References

Methods

__init__(energy_drive, energy_source, ...[, ...])	AbsHP components describe absorption heat pumps.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
set_efficiency_model(source_pinch, sink_pinch)	Defines the efficiency of the heat pump, i.e. the coefficient of performance (COP).

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
efficiency	Defines explicitly the efficiency of the heat pump, i.e. the coefficient of performance (COP).
energy_drive	Element enabling the pressurization of the heat pump refrigerant.
energy_sink	Element that receives thermal energy.
energy_source	Element that gives thermal energy.
given_sizing	The maximum capacity of the component, in kW.
name	str.
pump_consumption	
ref_production	Defines whether *energy_sink* or *energy_source* is the reference production.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.

- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property efficiency

Defines explicitly the efficiency of the heat pump, i.e. the coefficient of performance (COP).

The COP is a heating COP, i.e. it is used according to the two following constraints: 1. energy_from_source(t) + energy_from_drive(t) = energy_to_sink(t) 2. energy_from_sink(t) = energy_from_drive(t) * COP(t)

If called, replaces the definition of the efficiency using *set_efficiency_model* (default). int, float or numpy.ndarray

property energy_drive

Element enabling the pressurization of the heat pump refrigerant.

property energy_sink

Element that receives thermal energy. Must be warmed up if ThermalVectorPair. ——-

property energy_source

Element that gives thermal energy. Must be cooled down if ThermalVectorPair.

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property ref_production

Defines whether *energy_sink* or *energy_source* is the reference production. Either *energy_sink* or *energy_source*.

The reference production defines decision variable 'Q_P' and thus 'SP_P', which defines the cost of the component. If costs properties are typical values for a heat pump use, set

HP.ref_production=HP.energy_sink. If the costs properties are given for a chiller use, set *HP.ref_production=HP.energy_source*.

set_efficiency_model(*source_pinch*, *sink_pinch*, *as_HEx=True*, ***model_kwargs*)

Defines the efficiency of the heat pump, i.e. the coefficient of performance (COP).

The COP is a heating COP, i.e. it is used according to the two following constraints: 1. energy_from_source(t) + energy_from_drive(t) = energy_to_sink(t) 2. energy_from_sink(t) = energy_from_drive(t) * COP(t)

Parameters

- **source_pinch** (*int, float or numpy.ndarray*) – Temperature difference between *energy_source* and the refrigerant fluid of the heat pump (evaporator side). If *energy_source* is a ThermalVectorPair, the relevant temperature is the one of the cold vector (outcoming).
- **sink_pinch** (*int, float or numpy.ndarray*) – Temperature difference between *energy_sink* and the refrigerant fluid of the heat pump (condenser side). If *energy_sink* is a ThermalVectorPair, the relevant temperature is the one of the warm vector (outcoming).
- **as_HEx** (*bool, optional, default True*) – Describes the behavior of the component when the temperature of the heat source exceeds the one of the heat sink.
  - If True, the heat pump behaves similarly as a heat exchanger since its COP is very high. This high COP is calculated by defining equal temperatures for energy source and sink: new_source_temperature(t) = new_sink_temperature(t) = (source_temperature(t)+sink_temperature(t))/2
  - If False, COP model is applied as of, which could lead to non physical results.
- **model_kwargs** (*keyword arguments passed to the COP calculation function.*) –
  - For CompHP components, these can be:
    - 'fluid': {'ammonia', 'isobutane'}, default 'ammonia' Refrigerant fluid of the heat pump.
    - 'eta_is', float, default 0.75 Isentropic efficiency of the compression.
    - 'f_Q': float, default 0.2 Compressor heat loss ratio.
  - For AbsHP components, these can be:
    - 'couple': {'NH3/H2O', 'H2O/LiBr'}, default 'H2O/LiBr' Refrigerant and absorbent fluids.

Notes

See class *COPModels* from tamos.production.

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.2 tamos.production.BiomassBoiler

class tamos.production.**BiomassBoiler**(*energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

BiomassBoiler components produce heat given an energy efficiency of typical biomass boilers. This efficiency takes into account the condensing property of flue gases.

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is *energy_sink*.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_source** (FuelVector) – Biomass that is consumed.
- **energy_sink** (*ThermalVectorPair*) – Thermal flow that is warmed up by the boiler.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.

- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.

Methods

__init__(energy_source, energy_sink, properties)	BiomassBoiler components produce heat given an energy efficiency of typical biomass boilers.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
default_efficiency(T)	
set_efficiency_model(efficiency_function, pinch)	Defines the efficiency of the conversion of *energy_source* to *energy_sink* using a function of the cold temperature of *energy_sink*.

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
efficiency	Defines explicitly the efficiency of the conversion of *energy_source* to *energy_sink*.
energy_sink	Thermal flow that is warmed up by the boiler.
energy_source	
given_sizing	The maximum capacity of the component, in kW.
name	str.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.

- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property efficiency

Defines explicitly the efficiency of the conversion of *energy_source* to *energy_sink*. If called, replaces the definition of the efficiency using *set_efficiency_model* (default). int, float or numpy.ndarray

property energy_sink

Thermal flow that is warmed up by the boiler.

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

set_efficiency_model(*efficiency_function*, *pinch*)

Defines the efficiency of the conversion of *energy_source* to *energy_sink* using a function of the cold temperature of *energy_sink*.

Parameters

- **efficiency_function** (*callable f(T)*) – T is the temperature of the cold vector of *energy_sink*, in Kelvins (K).
- **pinch** (*int, float or numpy.ndarray*) – Temperature difference between the flue gases of the boiler and the cold vector of *energy_sink*, in Kelvins (K).

Notes

By default, set_efficiency_model is called with the *default_efficiency* attribute of this instance and pinch = 2.

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.3 tamos.production.CHP

class tamos.production.**CHP**(*energy_source*, *energy_sink_1*, *energy_sink_2*, *mode*, *properties*, *heat_efficiency_function*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*energy_source*, *energy_sink_1*, *energy_sink_2*, *mode*, *properties*, *heat_efficiency_function*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

CHP components transform a FuelVector element into heat and electricity.

This model is an adaptation of[1].

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = * F_P(e, t). For this component, e is *energy_sink_1*.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_source** (FuelVector) – Vector that is consumed.
- **energy_sink_1** (ElectricityVector) –
- **energy_sink_2** (*ThermalVectorPair*) – Thermal flow that is warmed up by the CHP.
- **mode** (*{'By-pass', 'Extraction-condensation', 'Back-pressure'}*) – Operating mode of the CHP:

[1] DAHL, Magnus, BRUN, Adam et ANDRESEN, Gorm B., 2019. Cost sensitivity of optimal sector-coupled district heating production systems. Energy. 1 janvier 2019. Vol.166, pp.624-636. DOI 10.1016/j.energy.2018.10.044.

  - 'Back-pressure': heat production is proportional to electricity production
  - 'Extraction-condensing': heat and electricity productions are decoupled, with electricity production being favored
  - 'By-pass': heat and electricity productions are decoupled, with heat production being favored
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "alpha": power-to-heat ratio (back-pressure line)
  - "beta": marginal power ratio (increase in power when heat decreases) (by-pass mode)
  - "ksi": marginal power ratio (increase in power when heat decreases) (extraction-condensation mode)
  - "eta": electrical efficiency
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **heat_efficiency_function** (*callable f(T)*) – [experimental] Only for modes 'Back-pressure' and 'Extraction-condensation'. T is the temperature of the cold vector of *energy_sink_2*, in Kelvins (K). See method *set_efficiency_model* of *GasBoiler* and *BiomassBoiler* classes. Setting *heat_efficiency=1* after instanciation makes this attribute unused.
- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.

References

Methods

__init__(energy_source, energy_sink_1, ...)	CHP components transform a FuelVector element into heat and electricity.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
set_heat_efficiency_model(...)	[experimental] Defines the heat efficiency of the CHP.

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
energy_sink_1	
energy_sink_2	
energy_source	Vector that is consumed.
given_sizing	The maximum capacity of the component, in kW.
heat_efficiency	[experimental] Defines explicitly the heat efficiency of the CHP.
mode	Operating mode of the CHP:
name	str.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*

- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- * *total_cost* = *CAPEX_share* + *OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property energy_source

Vector that is consumed.

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property heat_efficiency

[experimental] Defines explicitly the heat efficiency of the CHP. Only for modes 'Back-pressure' and 'Extraction-condensation'. If called, replaces the definition of the efficiency using *set_heat_efficiency_model* (default). Setting *heat_efficiency=1* is a safe value. int, float or numpy.ndarray

property mode

Operating mode of the CHP:

- 'Back-pressure': heat production is proportional to electricity production
- 'Extraction-condensing': heat and electricity productions are decoupled, with electricity production being favored
- 'By-pass': heat and electricity productions are decoupled, with heat production being favored

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

set_heat_efficiency_model(*heat_efficiency_function*, *pinch*)

[experimental] Defines the heat efficiency of the CHP.

Parameters

- **heat_efficiency_function** (*callable f(T)*) – T is the temperature of the cold vector of *energy_sink*, in Kelvins (K).
- **pinch** (*int, float or numpy.ndarray*) – Temperature difference between the flue gases of the boiler and the cold vector of *energy_sink*, in Kelvins (K).
- **classes.** (*See method set_efficiency_model of GasBoiler and BiomassBoiler*) –

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.4 tamos.production.COPModels

class tamos.production.**COPModels**

__init__()

Methods

__init__()	
absorption([couple])	Computes the expression of the COP model for absorption heat pumps.
compression([fluid, eta_is, f_Q])	Computes the expression of the COP model for compression heat pumps.
plot([range_T_H_O, range_T_C_O, ...])	Plots typical values returned by the absorption and compression COP models.

static absorption(*couple='H2O/LiBr'*)

Computes the expression of the COP model for absorption heat pumps.

This model is an adaptation of (Boudéhenn et al., 2016)[2].

Parameters

couple (*{'NH3/H2O', 'H2O/LiBr'}, default 'H2O/LiBr'*) – Refrigerant and absorbent fluids.

Returns

- T_H_O: temperature of the outcoming flow of energy_sink, in Kelvins (K)
- DT_H: difference of temperature between incoming and outcoming flows of energy_sink, in Kelvins (K)
- T_C_O: temperature of the outcoming flow of energy_source, in Kelvins (K)
- DT_C: difference of temperature between incoming and outcoming flows of energy_source, in Kelvins (K)
- DT_P_H: difference of temperature between the refrigerant fluid and energy_sink, in Kelvins (K)

[2] Boudéhenn F, Bonnot S, Demasles H, Lefrançois F, Perier-Muzet M, Triché D. Development and Performances Overview of Ammonia-water Absorption Chillers with Cooling Capacities from 5 to 100 kW. Energy Procedia 2016;91:707–16. https://doi.org/10.1016/j.egypro.2016.06.234.

- DT_P_C: difference of temperature between the refrigerant fluid and energy_source, in Kelvins (K)
- T_G_I: temperature of the incoming flow of energy_drive, in Kelvins (K)

Return type
A sympy callable returning the COP. Its arguments are

Notes

The COP is a heating COP, i.e. it is used according to the two following constraints: 1. energy_from_source(t) + energy_from_drive(t) = energy_to_sink(t) 2. energy_from_sink(t) = energy_from_drive(t) * COP(t)

References

static compression(*fluid='ammonia'*, *eta_is=0.75*, *f_Q=0.2*)

Computes the expression of the COP model for compression heat pumps.

This model is an adaptation of (Jensen et al., 2018)[1].

Parameters

- **fluid** (*{'ammonia', 'isobutane'}, optional, default 'ammonia'*) – Refrigerant fluid of the heat pump.
- **eta_is** (*float, default 0.75*) – Isentropic efficiency of the compression.
- **f_Q** (*float, default 0.2*) – Compressor heat loss ratio.

Returns

- T_H_O: temperature of the outcoming flow of energy_sink, in Kelvins (K)
- DT_H: difference of temperature between incoming and outcoming flows of energy_sink, in Kelvins (K)
- T_C_O: temperature of the outcoming flow of energy_source, in Kelvins (K)
- DT_C: difference of temperature between incoming and outcoming flows of energy_source, in Kelvins (K)
- DT_P_H: difference of temperature between the refrigerant fluid and energy_sink, in Kelvins (K)
- DT_P_C: difference of temperature between the refrigerant fluid and energy_source, in Kelvins (K)

Return type
A sympy callable returning the COP. Its arguments are

[1] JENSEN J. K, OMMEN T, REINHOLDT L, Et Al. Heat pump COP, part 2: generalized COP estimation of heat pump processes. 2018. https://doi.org/10.18462/IIR.GL.2018.1386.

Notes

The COP is a heating COP, i.e. it is used according to the two following constraints: 1. energy_from_source(t) + energy_from_drive(t) = energy_to_sink(t) 2. energy_from_sink(t) = energy_from_drive(t) * COP(t)

References

static plot(*range_T_H_O=range(278, 344)*, *range_T_C_O=range(273, 299, 5)*, *range_T_G_I=range(363, 403, 10)*, *celsius=False*)

Plots typical values returned by the absorption and compression COP models.

Parameters

- **range_T_H_O** (*list-like, optional, default range(5+273, 71+273, 1)*) – Temperatures of the outcoming flow of energy_sink, in Kelvins (K).
- **range_T_C_O** (*list-like, optional, default range(0+273, 26+273, 5)*) – Temperatures of the outcoming flow of energy_source, in Kelvins (K).
- **range_T_G_I** (*list-like, optional, default range(90+273, 130+273, 10)*) – Temperatures of the incoming flow of energy_drive, in Kelvins (K). Relevant for absorption COP model.
- **celsius** (*bool, optional, default False*) – If True, arguments *range_T_H_O*, *range_T_C_O* and *range_T_G_I* must describe temperatures in Celsius (°C).

Returns
figs

Return type
list of plotly.graph_objs._figure.Figure

Notes

Figures are displayed on the local host browser using the plotly package.

3.5 tamos.production.CompHP

class tamos.production.**CompHP**(*energy_drive*, *energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *name=None*, *eco_count=True*, *units_number_ub=1*, *units_number_lb=1*)

__init__(*energy_drive*, *energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *name=None*, *eco_count=True*, *units_number_ub=1*, *units_number_lb=1*)

CompHP components describe vapor-compression heat pumps. The COP model is an adaptation of (Jensen et al., 2018)[1].

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.

[1] JENSEN J. K, OMMEN T, REINHOLDT L, Et Al. Heat pump COP, part 2: generalized COP estimation of heat pump processes. 2018. https://doi.org/10.18462/IIR.GL.2018.1386.

- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is either *energy_sink* or *energy_source* depending on the *ref_production* attribute.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_drive** (ElectricityVector) – Electricity consumed by the machine.
- **energy_source** (*ThermalVectorPair*, ThermalVector) – Element that gives thermal energy. Must be cooled down if ThermalVectorPair.
- **energy_sink** (*ThermalVectorPair*, ThermalVector) – Element that receives thermal energy. Must be warmed up if ThermalVectorPair.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.

References

Methods

__init__(energy_drive, energy_source, ...[, ...])	CompHP components describe vapor-compression heat pumps.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
set_efficiency_model(source_pinch, sink_pinch)	Defines the efficiency of the heat pump, i.e. the coefficient of performance (COP).

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
efficiency	Defines explicitly the efficiency of the heat pump, i.e. the coefficient of performance (COP).
energy_drive	Element enabling the pressurization of the heat pump refrigerant.
energy_sink	Element that receives thermal energy.
energy_source	Element that gives thermal energy.
given_sizing	The maximum capacity of the component, in kW.
name	str.
ref_production	Defines whether *energy_sink* or *energy_source* is the reference production.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*

- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- * *total_cost* = *CAPEX_share* + *OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property efficiency

Defines explicitly the efficiency of the heat pump, i.e. the coefficient of performance (COP).

The COP is a heating COP, i.e. it is used according to the two following constraints: 1. energy_from_source(t) + energy_from_drive(t) = energy_to_sink(t) 2. energy_from_sink(t) = energy_from_drive(t) * COP(t)

If called, replaces the definition of the efficiency using *set_efficiency_model* (default). int, float or numpy.ndarray

property energy_drive

Element enabling the pressurization of the heat pump refrigerant.

property energy_sink

Element that receives thermal energy. Must be warmed up if ThermalVectorPair. ——-

property energy_source

Element that gives thermal energy. Must be cooled down if ThermalVectorPair.

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property ref_production

Defines whether *energy_sink* or *energy_source* is the reference production. Either *energy_sink* or *energy_source*.

The reference production defines decision variable 'Q_P' and thus 'SP_P', which defines the cost of the component. If costs properties are typical values for a heat pump use, set *HP.ref_production=HP.energy_sink*. If the costs properties are given for a chiller use, set *HP.ref_production=HP.energy_source*.

set_efficiency_model(*source_pinch*, *sink_pinch*, *as_HEx=True*, ***model_kwargs*)

Defines the efficiency of the heat pump, i.e. the coefficient of performance (COP).

The COP is a heating COP, i.e. it is used according to the two following constraints: 1. energy_from_source(t) + energy_from_drive(t) = energy_to_sink(t) 2. energy_from_sink(t) = energy_from_drive(t) * COP(t)

Parameters

- **source_pinch** (*int, float or numpy.ndarray*) – Temperature difference between *energy_source* and the refrigerant fluid of the heat pump (evaporator side). If *energy_source* is a ThermalVectorPair, the relevant temperature is the one of the cold vector (outcoming).
- **sink_pinch** (*int, float or numpy.ndarray*) – Temperature difference between *energy_sink* and the refrigerant fluid of the heat pump (condenser side). If *energy_sink* is a ThermalVectorPair, the relevant temperature is the one of the warm vector (outcoming).
- **as_HEx** (*bool, optional, default True*) – Describes the behavior of the component when the temperature of the heat source exceeds the one of the heat sink.
  - If True, the heat pump behaves similarly as a heat exchanger since its COP is very high. This high COP is calculated by defining equal temperatures for energy source and sink: new_source_temperature(t) = new_sink_temperature(t) = (source_temperature(t)+sink_temperature(t))/2
  - If False, COP model is applied as of, which could lead to non physical results.
- **model_kwargs** (*keyword arguments passed to the COP calculation function.*) –
  - For CompHP components, these can be:
    - 'fluid': {'ammonia', 'isobutane'}, default 'ammonia' Refrigerant fluid of the heat pump.
    - 'eta_is', float, default 0.75 Isentropic efficiency of the compression.
    - 'f_Q': float, default 0.2 Compressor heat loss ratio.
  - For AbsHP components, these can be:
    - 'couple': {'NH3/H2O', 'H2O/LiBr'}, default 'H2O/LiBr' Refrigerant and absorbent fluids.

Notes

See class *COPModels* from tamos.production.

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.6 tamos.production.DryCooler

class tamos.production.**DryCooler**(*energy_drive*, *energy_source*, *energy_sink*, *properties*, *efficiency=36.36363636363637*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*energy_drive*, *energy_source*, *energy_sink*, *properties*, *efficiency=36.36363636363637*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

DryCooler components dissipate thermal energy using fans having an electrical consumption.

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is *energy_source*.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_drive** (ElectricityVector) – Electricty used to facilitate energy transfer (e.g. using fans).
- **energy_source** (*ThermalVectorPair*, ThermalVector) – Element which must be dissipated. Must be cooled down if ThermalVectorPair.
- **energy_sink** (*ThermalVectorPair*, ThermalVector) – Element receiving thermal energy of the dissipation of *energy_source*. Must be warmed up if ThermalVectorPair.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **efficiency** (*int, float or numpy.ndarray, optional, default 1/0.0275*) – Number of units of dissipated heat per unit of electricity.
- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –

- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.

Notes

Flows of *energy_source* and *energy_sink* are equal.

Examples

```
>>> air = ThermalVector(temperature = 273 + 20, name="Ambiant air")
>>> excess_heat = ThermalVectorPair(in_TV=in_TV, out_TV=out_TV)
>>> excess_heat.is_cooled
    True
>>> DryCooler(electricity, excess_heat, air, properties, efficiency = 20)
```

Heat from *excess_heat* is converted (i.e. dissipated) to *air* with an electricity consumption being 20 times smaller than the converted thermal power.

Methods

__init__(energy_drive, energy_source, ...[, ...])	DryCooler components dissipate thermal energy using fans having an electrical consumption.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
efficiency	Number of units of dissipated heat per unit of electricity.
energy_drive	Electricty used to facilitate energy transfer (e.g.
energy_sink	Element receiving thermal energy of the dissipation of *energy_source*.
energy_source	Element which must be dissipated.
given_sizing	The maximum capacity of the component, in kW.
name	str.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- * *total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property efficiency

Number of units of dissipated heat per unit of electricity. int, float or numpy.ndarray

property energy_drive

Electricty used to facilitate energy transfer (e.g. using fans).

property energy_sink

Element receiving thermal energy of the dissipation of *energy_source*. Must be warmed up if ThermalVectorPair. ThermalVectorPair, ThermalVector

property energy_source

Element which must be dissipated. Must be cooled down if ThermalVectorPair. ThermalVectorPair, ThermalVector

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.7 tamos.production.ElectricHeater

class tamos.production.**ElectricHeater**(*energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

ElectricHeater components produce heat following a unit efficiency.

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is *energy_sink*.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_source** (ElectricityVector) –
- **energy_sink** (*ThermalVectorPair*) – Thermal flow that is warmed up by the boiler.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.

Methods

__init__(energy_source, energy_sink, properties)	ElectricHeater components produce heat following a unit efficiency.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
set_efficiency_model(efficiency_function, pinch)	Defines the efficiency of the conversion of *energy_source* to *energy_sink* using a function of the cold temperature of *energy_sink*.

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
efficiency	Defines explicitly the efficiency of the conversion of *energy_source* to *energy_sink*.
energy_sink	Thermal flow that is warmed up by the boiler.
energy_source	
given_sizing	The maximum capacity of the component, in kW.
name	str.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property efficiency

Defines explicitly the efficiency of the conversion of *energy_source* to *energy_sink*. If called, replaces the definition of the efficiency using *set_efficiency_model* (default). int, float or numpy.ndarray

property energy_sink

Thermal flow that is warmed up by the boiler.

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

set_efficiency_model(*efficiency_function*, *pinch*)

Defines the efficiency of the conversion of *energy_source* to *energy_sink* using a function of the cold temperature of *energy_sink*.

Parameters

- **efficiency_function** (*callable f(T)*) – T is the temperature of the cold vector of *energy_sink*, in Kelvins (K).
- **pinch** (*int, float or numpy.ndarray*) – Temperature difference between the flue gases of the boiler and the cold vector of *energy_sink*, in Kelvins (K).

Notes

By default, set_efficiency_model is called with the *default_efficiency* attribute of this instance and pinch = 2.

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.8 tamos.production.ElementConverter

class tamos.production.**ElementConverter**(*element_1*, *element_2*, *direction*, *name=None*)

__init__(*element_1*, *element_2*, *direction*, *name=None*)

ElementConverter components convert an element into another.

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is either *element_1* or *element_2* depending on the *direction* attribute.

This component does not declare any KPI.

Parameters

- **element_1** (ElectricityVector, FuelVector, *ThermalVectorPair*, ThermalVector) –
- **element_2** (ElectricityVector, FuelVector, *ThermalVectorPair*, ThermalVector) –
- **direction** (*{'produced', 'consumed', 'both'}*) – Related to *element_1*.
  - 'produced': a flow of *element_1* is produced, a flow of *element_2* is consumed
  - 'consumed': a flow of *element_2* is produced, a flow of *element_1* is consumed
  - **'both': depending on the time step t, a flow of *element_1* is produced or consumed**
    In this mode, decision variable 'Q_P' has no upper bound.
- **name** (*str, optional*) –

Notes

No *properties* argument must be specified for this component:

- "LB max output power (kW)" is set to 0 kW
- "UB max output power (kW)" is set to 1 GW
- no costs are associated with the component

Methods

__init__(element_1, element_2, direction[, name])	ElementConverter components convert an element into another.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

direction	Related to *element_1*.
eco_count	Whether this instance contributes to the system "Eco" KPI.
element_1	ElectricityVector, FuelVector, ThermalVectorPair, ThermalVector
element_2	ElectricityVector, FuelVector, ThermalVectorPair, ThermalVector
given_sizing	The maximum capacity of the component, in kW.
name	str.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property direction

Related to *element_1*.

- 'produced': a flow of *element_1* is produced, a flow of *element_2* is consumed
- 'consumed': a flow of *element_2* is produced, a flow of *element_1* is consumed
- **'both': depending on the time step t, a flow of *element_1* is produced or consumed**
  In this mode, decision variable 'Q_P' has no upper bound.

{'produced', 'consumed', 'both'}

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property element_1

ElectricityVector, FuelVector, ThermalVectorPair, ThermalVector

property element_2

ElectricityVector, FuelVector, ThermalVectorPair, ThermalVector

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.9 tamos.production.FPSolar

class tamos.production.**FPSolar**(*energy_sink*, *air_temperature*, *total_irradiance*, *properties*, *pinch=3*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*energy_sink*, *air_temperature*, *total_irradiance*, *properties*, *pinch=3*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

FPSolar components convert solar irradiance to heat.

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is *energy_sink*.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_sink** (*ThermalVectorPair*) – Thermal flow that is warmed up.
- **air_temperature** (*int, float or numpy.ndarray*) – In Kelvins (K). Temperature of the air surrounding the solar panels.
- **total_irradiance** (*int, float or numpy.ndarray*) – In kW/m^2. Solar irradiance received on the normal of the solar panels.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/m2)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
  - "eta0"
  - "a1 (W/(m2.K))"
  - "a2 (W/(m2.K2))"
  - "a5 (J/(m2.K))"
  - "UB area (m2)"
- **pinch** (*int, float or numpy.ndarray, optional, default 3*) – Difference between the temperature of the fluid circulating in the solar panels and the one of *energy_sink*.

- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.

Notes

1. Parameters "eta0", "a1 (W/(m2.K))", "a2 (W/(m2.K2))" and "a5 (J/(m2.K))" can be found in databases like the Solar Keymark database.
2. 'FPSolar' stands for 'Flat plate solar thermal'.

Methods

__init__(energy_sink, air_temperature, ...)	FPSolar components convert solar irradiance to heat.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

air_temperature	Temperature of the air surrounding the solar panels.
eco_count	Whether this instance contributes to the system "Eco" KPI.
energy_sink	Thermal flow that is warmed up.
given_sizing	The maximum capacity of the component, in kW.
name	str.
pinch	Difference between the temperature of the fluid circulating in the solar panels and the one of *energy_sink*.
total_irradiance	Solar irradiance received on the normal of the solar panels.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

property air_temperature

Temperature of the air surrounding the solar panels. In Kelvins (K). int, float or numpy.ndarray

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- * *total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property energy_sink

Thermal flow that is warmed up. ThermalVectorPair

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property pinch

Difference between the temperature of the fluid circulating in the solar panels and the one of *energy_sink*. int, float or numpy.ndarray

property total_irradiance

Solar irradiance received on the normal of the solar panels. In kW/m^2. int, float or numpy.ndarray

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.10 tamos.production.GasBoiler

class tamos.production.**GasBoiler**(*energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

GasBoiler components produce heat given an energy efficiency of typical gas boilers. This efficiency takes into account the condensing property of flue gases.

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is *energy_sink*.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_source** (FuelVector) – Natural gas that is consumed.
- **energy_sink** (*ThermalVectorPair*) – Thermal flow that is warmed up by the boiler.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"

- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.

Methods

__init__(energy_source, energy_sink, properties)	GasBoiler components produce heat given an energy efficiency of typical gas boilers.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
default_efficiency(T)	
set_efficiency_model(efficiency_function, pinch)	Defines the efficiency of the conversion of *energy_source* to *energy_sink* using a function of the cold temperature of *energy_sink*.

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
efficiency	Defines explicitly the efficiency of the conversion of *energy_source* to *energy_sink*.
energy_sink	Thermal flow that is warmed up by the boiler.
energy_source	
given_sizing	The maximum capacity of the component, in kW.
name	str.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property efficiency

Defines explicitly the efficiency of the conversion of *energy_source* to *energy_sink*. If called, replaces the definition of the efficiency using *set_efficiency_model* (default). int, float or numpy.ndarray

property energy_sink

Thermal flow that is warmed up by the boiler.

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

set_efficiency_model(*efficiency_function*, *pinch*)

Defines the efficiency of the conversion of *energy_source* to *energy_sink* using a function of the cold temperature of *energy_sink*.

Parameters

- **efficiency_function** (*callable f(T)*) – T is the temperature of the cold vector of *energy_sink*, in Kelvins (K).
- **pinch** (*int, float or numpy.ndarray*) – Temperature difference between the flue gases of the boiler and the cold vector of *energy_sink*, in Kelvins (K).

Notes

By default, set_efficiency_model is called with the *default_efficiency* attribute of this instance and pinch = 2.

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.11 tamos.production.HeatExchanger

class tamos.production.**HeatExchanger**(*energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *efficiency=0.95*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*energy_source*, *energy_sink*, *properties*, *given_sizing=None*, *efficiency=0.95*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

HeatExchanger components convert heat in a passive way.

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is *energy_sink*.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **energy_source** (*ThermalVectorPair*, ThermalVector) – Element that gives thermal energy. Must be cooled down if ThermalVectorPair.
- **energy_sink** (*ThermalVectorPair*, ThermalVector) – Element that receives thermal energy. Must be warmed up if ThermalVectorPair.

- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **efficiency** (*int, float or numpy.ndarray, optional, default 0.95*) – The amount of energy received by *energy_sink* for a unit of energy from *energy_source*.
- **name** (*str, optional*) –
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max output power (kW)" property is different from 0.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.

Methods

__init__(energy_source, energy_sink, properties)	HeatExchanger components convert heat in a passive way.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
efficiency	The amount of energy received by *energy_sink* for a unit of energy from *energy_source*.
energy_sink	Element that receives thermal energy.
energy_source	Element that gives thermal energy.
given_sizing	The maximum capacity of the component, in kW.
name	str.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property efficiency

The amount of energy received by *energy_sink* for a unit of energy from *energy_source*. int, float or numpy.ndarray

property energy_sink

Element that receives thermal energy. Must be warmed up if ThermalVectorPair. ThermalVectorPair, ThermalVector

property energy_source

Element that gives thermal energy. Must be cooled down if ThermalVectorPair. ThermalVectorPair, ThermalVector

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

3.12 tamos.production.Pump

class tamos.production.**Pump**(*element_1*, *element_2*, *energy_drive*, *properties*, *direction='both'*, *pump_consumption=0.005*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*element_1*, *element_2*, *energy_drive*, *properties*, *direction='both'*, *pump_consumption=0.005*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

Pump components convert a ThermalVectorPair into another ThermalVectorPair. Electricity is consumed in proportion of the pumped power.

This component declares the following exported decision variables:

- X_P, binary. Whether the component is used by the hub.
- SP_P, continuous, in kW. The maximum capacity of the component. Defines the investment costs.
- For all t, for all element e, F_P(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).
- For all t, Q_P(t), continuous, in kW. The reference power related to the component. Defines the variable cost. This power is a lower bound of SP_P. There exists one element e such that Q_P(t) = F_P(e, t) or Q_P(t) = - F_P(e, t). For this component, e is *energy_drive*.

This component declares the following KPIs:

- *COST_production* In euros. Contributes to the "Eco" objective function.

Parameters

- **element_1** (*ThermalVectorPair*) –
- **element_2** (*ThermalVectorPair*) –
- **energy_drive** (ElectricityVector) –
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max output power (kW)"
  - "UB max output power (kW)"
  - "CAPEX (EUR/kW)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **direction** (*{'produced', 'consumed', 'both'}, optional, default 'both'*) – Related to *element_1*.
  - 'produced': a flow of *element_1* is produced, a flow of *element_2* is consumed
  - 'consumed': a flow of *element_2* is produced, a flow of *element_1* is consumed
  - **'both': depending on the time step t, a flow of *element_1* is produced or consumed**
    In this mode, decision variable 'Q_P' has no upper bound. In 'Eco' optimization, the component cost constrains *Q_P*.
- **pump_consumption** (*int, float or numpy.ndarray, optional, default 0.005*) – Instantaneous electrical consumption of the component in proportion of the pumped power.
- **name** (*str, optional*) –

Methods

__init__(element_1, element_2, energy_drive, ...)	Pump components convert a ThermalVectorPair into another ThermalVectorPair.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

direction	Related to *element_1*.
eco_count	Whether this instance contributes to the system "Eco" KPI.
element_1	ThermalVectorPair
element_2	ThermalVectorPair
energy_drive	ElectricityVector
given_sizing	The maximum capacity of the component, in kW.
name	str.
pump_consumption	
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property direction

Related to *element_1*.

- 'produced': a flow of *element_1* is produced, a flow of *element_2* is consumed
- 'consumed': a flow of *element_2* is produced, a flow of *element_1* is consumed
- **'both': depending on the time step t, a flow of *element_1* is produced or consumed**
  In this mode, decision variable 'Q_P' has no upper bound.

{'produced', 'consumed', 'both'}

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property element_1

ThermalVectorPair

property element_2

ThermalVectorPair

property energy_drive

ElectricityVector

property given_sizing

The maximum capacity of the component, in kW. Relates to decision variable 'SP_P'. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

CHAPTER

FOUR

STORAGE COMPONENTS

tamos.storage.OneVector	(vector, properties)
tamos.storage.Thermocline	(stored_TVP[, ...])

4.1 tamos.storage.OneVector

class tamos.storage.**OneVector**(*vector*, *properties*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

__init__(*vector*, *properties*, *given_sizing=None*, *name=None*, *units_number_ub=1*, *units_number_lb=1*, *eco_count=True*)

OneVector components store Electricity or FuelVector elements.

This component declares the following exported decision variables:

- X_S, binary. Whether the component is used by the hub.
- SE_S, continuous, in kWh. The maximum energetical capacity of the component.
- For all t, E_S(t), continuous, in kWh. State of charge of the storage.
- For all t, for all element e, F_S(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).

This component declares the following KPIs:

- *COST_storage* In euros. Contributes to the "Eco" objective function.

Parameters

- **vector** (*ElectricityVector or FuelVector*) – Stored element.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component. The *properties* attribute must include the following keys:
  - "LB max energy (kWh)"
  - "UB max energy (kWh)"
  - "Charge/discharge delay (h)"
  - "Energy conservation (/h)"
  - "CAPEX energy (EUR/kWh)"

– ”OPEX energy (%CAPEX)”

- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kWh. Relates to decision variable ‘SE_S’. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if “LB max energy (kWh)” property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if “LB max energy (kWh)” property is different from 0.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system “Eco” KPI.

Methods

__init__(vector, properties[, given_sizing, ...])	OneVector components store Electricity or FuelVector elements.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
given_sizing	The maximum capacity of the component, in kg (Thermocline) or kWh (OneVector).
name	str.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.
vector	Stored element

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its ‘Lifetime’ and ‘Discount rate (%)’ properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.

- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property given_sizing

The maximum capacity of the component, in kg (Thermocline) or kWh (OneVector). Relates to decision variable 'SE_S'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

property vector

Stored element

4.2 tamos.storage.Thermocline

class tamos.storage.**Thermocline**(*stored_TVP*, *properties=None*, *given_sizing=None*, *units_number_ub=1*, *units_number_lb=1*, *name=None*, *eco_count=True*)

__init__(*stored_TVP*, *properties=None*, *given_sizing=None*, *units_number_ub=1*, *units_number_lb=1*, *name=None*, *eco_count=True*)

Thermocline components model a perfectly stratified thermal energy storage.

This component declares the following exported decision variables:

- X_S, binary. Whether the component is used by the hub.
- SE_S, continuous, in kg. The maximum energetical capacity of the component.
- For all t, E_S(t), continuous, in kg. State of charge of the storage.
- For all t, for all element e, F_S(e, t), continuous, in kW. The power related to element e entering the component (i.e. leaving the hub interface).

This component declares the following KPIs:

- *COST_storage* In euros. Contributes to the "Eco" objective function.

Parameters

- **stored_TVP** (*ThermalVectorPair*) – Stored element. Storage is full when its entire content is the incoming vector of *stored_TVP*. Please see note 2) for an explanation of the restriction regarding *stored_TVP*.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the component.

  The *properties* attribute must include the following keys:

  – "LB max energy (kg)"

  – "UB max energy (kg)"

  – "Charge/discharge delay (h)"

  – "Energy conservation (/h)"

  – "CAPEX energy (EUR/kg)"

  – "OPEX energy (%CAPEX)"
- **given_sizing** (*int or float, optional*) – The maximum capacity of the component, in kg. Relates to decision variable 'SE_S'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed.
- **name** (*str, optional*) –
- **units_number_lb** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max energy (kg)" property is different from 0.
- **units_number_ub** (*int, optional, default 1*) – The lower bound (upper bound) of the number of real components that this instance aims to stand for. Setting *units_number_lb* (*units_number_ub*) has a meaning if "LB max energy (kg)" property is different from 0.

- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.

Notes

1. This modelisation is equivalent to 2 OneVector storages with the constraint state_of_charge_1 + state_of_charge_2 = state_of_charge (i.e.: the discharge of one side leads to the charge of the other side).
2. The instantaneous energetical state of charge of the storage depends on the product m(t) x Cp(t) x DT(t) where:
   - m(t) is the instantaneous mass state of charge of the storage (decision variable: E_S)
   - Cp(t) is the specific heat capacity of *stored_TVP*
   - DT(t) is the temperature difference between incoming and outcoming flows of *stored_TVP*

   Thus, in the general case, the storage can be charged when Cp(t) x DT(t) is low (requiring low charging power, decision variable: F_S) and discharged when Cp(t) x DT(t) is high; which has no physical meaning. For this reason, the Thermocline component must not be used with a ThermalVectorPair *stored_TVP* having a variable Cp(t) x DT(t) product.

Methods

__init__(stored_TVP[, properties, ...])	Thermocline components model a perfectly stratified thermal energy storage.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

eco_count	Whether this instance contributes to the system "Eco" KPI.
given_sizing	The maximum capacity of the component, in kg (Thermocline) or kWh (OneVector).
name	str.
stored_TVP	Stored element.
units_number_lb	The lower bound of the number of real components that this instance aims to stand for.
units_number_ub	The upper bound of the number of real components that this instance aims to stand for.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.

- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- * *total_cost* = *CAPEX_share* + *OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI. bool

property given_sizing

The maximum capacity of the component, in kg (Thermocline) or kWh (OneVector). Relates to decision variable 'SE_S'. If specified, only the operation of this component is performed by the MILP solver. If let unknown, both sizing and operation are performed. int or float

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property stored_TVP

Stored element. Storage is full when its entire content is the incoming vector of *stored_TVP*.

property units_number_lb

The lower bound of the number of real components that this instance aims to stand for. Setting *units_number_lb* has a meaning if "LB max output power (kW)" property is different from 0. int

property units_number_ub

The upper bound of the number of real components that this instance aims to stand for. Setting *units_number_ub* has a meaning if "LB max output power (kW)" property is different from 0. int

property used_elements

Elements used by the component.

CHAPTER

FIVE

ELEMENTIO COMPONENTS

tamos.elementIO.Cost([cost, carbon_cost, name])	
tamos.elementIO.Grid(element[, emissions, ...])	
tamos.elementIO.Load(load, element[, ...])	

5.1 tamos.elementIO.Cost

class tamos.elementIO.**Cost**(*cost=None*, *carbon_cost=None*, *name=None*)

__init__(*cost=None*, *carbon_cost=None*, *name=None*)

Defines the cost of buying energy from or selling energy to the outside of the energy system. A Cost instance must be passed to a Grid or Load instance to be effective. All costs are defined relatively to the flow of the *element* attribute of the Grid or Load instance.

Cost instances declare the following KPIs:

- *COST_element* In euros. Contributes to the "Eco" objective function. Related to the *cost* attribute.
- *Carbon tax* In euros. Contributes to the "Eco" objective function. Related to the *carbon_cost* attribute.

Parameters

- **cost** (*int, float or numpy.ndarray, optional*) – Cost associated with a positive flow of *element*. In euros/kWh. Usually negative.
- **carbon_cost** (*int, float or numpy.ndarray, optional*) – Cost associated with positive CO2 emissions regarding the flow of 'element'. In euros/kgEqCO2. Usually positive.
- **name** (*str, optional*) –

Examples

```
>>> grid=  Grid(element=electricity, emissions=0.3)
>>> cost = Cost(cost=-0.2, carbon_cost=0.1)
>>> grid.element_cost = cost
```

The energy system paies 0.2€ to buy 1 kWh of *electricity* from *grid*, added to 0.3 * 0.1 = 0.03€ paid as a carbon tax.

Methods

__init__([cost, carbon_cost, name])	Defines the cost of buying energy from or selling energy to the outside of the energy system.

Attributes

carbon_cost	Cost associated with positive CO2 emissions regarding the flow of 'element'.
cost	Cost associated with a positive flow of *element*.
name	str.

property carbon_cost

Cost associated with positive CO2 emissions regarding the flow of 'element'. In euros/kgEqCO2. Usually positive. int, float or numpy.ndarray

property cost

Cost associated with a positive flow of *element*. In euros/kWh. Usually negative. int, float or numpy.ndarray

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ‘ ’ { } ~.

Type
Name of the instance

5.2 tamos.elementIO.Grid

class tamos.elementIO.**Grid**(*element*, *emissions=None*, *element_cost=None*, *exergy_count=True*, *name=None*)

__init__(*element*, *emissions=None*, *element_cost=None*, *exergy_count=True*, *name=None*)

Allows unconstrained element exchanges between the energy system and its environment.

Grid components are associated with the following exported decision variables:

- X_EXT, binary. Whether the Grid instance is used by the hub.
- For all t, F_EXT(t), continuous, in kW. The power related to *element* entering the grid (i.e. leaving the hub interface).

Grid components declare the following KPIs:

- *ElementIO CO2* In kgEqCO2. Defines the "CO2" objective function. Related to the *emissions* attribute.
- *ElementIO Exergy* In kWh. Defines the "Exergy" objective function. Related to the *exergy_factor* attribute of *element*.

Parameters

- **element** (*ElectricityVector, FuelVector, ThermalVector or ThermalVectorPair*) –
- **emissions** (*int, float or numpy.ndarray, optional*) – Quantity of CO2 associated with a positive flow of *element*. In kgEqCO2/kWh. Usually negative. If None, emissions are not accounted for.
- **element_cost** (*Cost, optional*) – Cost associated with a positive flow of 'element'. If None, no costs are taken into account.
- **exergy_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Exergy" KPI.
- **name** (*str, optional*) –

Methods

__init__(element[, emissions, element_cost, ...])	Allows unconstrained element exchanges between the energy system and its environment.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

element	Element exchanged between the hub interface and this instance.
element_cost	Cost instance associated with a positive flow of 'element'.
emissions	Quantity of CO2 associated with a positive power flow of *element*.
exergy_count	Whether this instance contributes to the system "Exergy" KPI.
name	str.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.

- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- * *total_cost* = *CAPEX_share* + *OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property element

Element exchanged between the hub interface and this instance.

property element_cost

Cost instance associated with a positive flow of 'element'. Cost instance

property emissions

Quantity of CO2 associated with a positive power flow of *element*. In kgEqCO2/kWh. Usually negative. int, float or numpy.ndarray

Examples

Examples below are for a Grid component, but Load components behaves similarly.

```
>>> natural_gas_grid = Grid(element=natural_gas)
>>> natural_gas_grid.emissions = - 0.25
```

When the Grid component *natural_gas_grid* receives 1 kWh of natural gas from the hub interface the net CO2 emissions of the energy system decrease of 0.250 kgEqCO2. Conversely, if the energy_system receives 1 kWh of natural gas from *natural_gas_grid*, its CO2 emissions increase of 0.250 kgEqCO2.

```
>>> thermal_grid = Grid(element=thermal_source)
>>> thermal_grid.emissions = - 0.05
```

thermal_grid is such that when the power flow of *thermal_source* is 1 kWh (positive):

- thermal_source.in_TV enters the grid (i.e leaves the energy system) and thermal_source.out_TV leaves the grid (i.e enters the energy_system)
- the net CO2 emissions of the energy system decrease of 50 gEqCO2.

property exergy_count

Whether this instance contributes to the system "Exergy" KPI. bool

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property used_elements

Elements used by the component.

5.3 tamos.elementIO.Load

class tamos.elementIO.**Load**(*load*, *element*, *emissions=None*, *element_cost=None*, *exergy_count=True*, *name=None*)

__init__(*load*, *element*, *emissions=None*, *element_cost=None*, *exergy_count=True*, *name=None*)

Allows constrained element exchanges between the energy system and its environment.

Load components are associated with the following exported decision variables:

- X_EXT, binary. Whether the Load instance is used by the hub. To force the use of this instance by a hub, set components_assemblies at Hub or MILPModel level. For instance:

```
>>> hub.components_assemblies = [(1, 1, heating_load)]
```

- For all t, F_EXT(t), continuous, in kW. The power related to *element* entering the load (i.e. leaving the hub).

Load components declare the following KPIs:

- ***ElementIO CO2***
  In kgEqCO2. Defines the "CO2" objective function. Related to the *emissions* attribute.
- ***ElementIO Exergy***
  In kWh. Defines the "Exergy" objective function. Related to the *exergy_factor* attribute of *element*.

Parameters

- **load** (*int, float or numpy.ndarray*) – The power pattern that constrains element exchanges. In kW. Same sign than the flow of 'element'.
- **element** (ElectricityVector, FuelVector, ThermalVector *or* *ThermalVectorPair*) –
- **emissions** (*int, float or numpy.ndarray, optional*) – Quantity of CO2 associated with a positive flow of *element*. In kgEqCO2/kWh. Usually negative. If None, emissions are not accounted for.
- **element_cost** (Cost, *optional*) – Cost associated with a positive flow of 'element'. If None, no costs are taken into account.
- **exergy_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Exergy" KPI.
- **name** (*str, optional*) –

Notes

The Load class defines a multiplication operation to speed up the definition of several Load instances. Multiplication makes a copy of *load* attribute, so that mutable numpy.ndarray are made independent. For instance:

```
>>> load_1 = Load(load=-4, element=element1, emissions=-0.2, exergy_
↪count=False)
>>> load_2 = 3 * load_1
>>> load_2.load
 -12
>>> load_2.element is element1
 True
```

Examples

```
>>> Load(load=10, element=electricity)
```

A constant electrical power of 10 kW will leave the hub interface.

```
>>> thermal_source = fetch_TVP(in_TV=in_TV, out_TV=out_TV)
>>> load_1 = Load(load=-5, element=thermal_source)
>>> load_2 = Load(load=5, element=~thermal_source)
```

In both cases of load_1 and load_2, a constant power of 5 kW is exchanged between the hub interface and the Load component, with *out_TV* entering the hub interface (i.e. leaving the Load component).

Methods

__init__(load, element[, emissions, ...])	Allows constrained element exchanges between the energy system and its environment.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Attributes

element	Element exchanged between the hub interface and this instance.
element_cost	Cost instance associated with a positive flow of 'element'.
emissions	Quantity of CO2 associated with a positive power flow of *element*.
exergy_count	Whether this instance contributes to the system "Exergy" KPI.
load	The power pattern that constrains element exchanges.
name	str.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property element

Element exchanged between the hub interface and this instance.

property element_cost

Cost instance associated with a positive flow of 'element'. Cost instance

property emissions

Quantity of CO2 associated with a positive power flow of *element*. In kgEqCO2/kWh. Usually negative. int, float or numpy.ndarray

Examples

Examples below are for a Grid component, but Load components behaves similarly.

```
>>> natural_gas_grid = Grid(element=natural_gas)
>>> natural_gas_grid.emissions = - 0.25
```

When the Grid component *natural_gas_grid* receives 1 kWh of natural gas from the hub interface the net CO2 emissions of the energy system decrease of 0.250 kgEqCO2. Conversely, if the energy_system receives 1 kWh of natural gas from *natural_gas_grid*, its CO2 emissions increase of 0.250 kgEqCO2.

```
>>> thermal_grid = Grid(element=thermal_source)
>>> thermal_grid.emissions = - 0.05
```

thermal_grid is such that when the power flow of *thermal_source* is 1 kWh (positive):

- thermal_source.in_TV enters the grid (i.e leaves the energy system) and thermal_source.out_TV leaves the grid (i.e enters the energy_system)
- the net CO2 emissions of the energy system decrease of 50 gEqCO2.

property exergy_count

Whether this instance contributes to the system "Exergy" KPI. bool

property load

The power pattern that constrains element exchanges. In kW. Same sign than the flow of 'element'. int, float or numpy.ndarray

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

property used_elements

Elements used by the component.

CHAPTER

SIX

NETWORK COMPONENTS

6.1 Components

tamos.network.NonThermalNetwork(...[, ...])	
tamos.network.ThermalNetwork(hubs_locations, ...)	
tamos.network.HREThermalNetwork(...[, ...])	

6.1.1 tamos.network.NonThermalNetwork

class tamos.network.**NonThermalNetwork**(*hubs_locations*, *element*, *properties*, *production_hub*, *scale_factor=1*, *eco_count=True*, *name=None*)

__init__(*hubs_locations*, *element*, *properties*, *production_hub*, *scale_factor=1*, *eco_count=True*, *name=None*)

NonThermalNetwork instances makes possible to share ElectricityVector or FuelVector elements between hubs.

Power is exchanged between two hubs, given a distribution loss related to the "Losses (%/km)" property. All distribution losses must be compensated for by an additional power in *production_hub*.

NonThermalNetwork components are associated with the following exported decision variables:

- X_N(hub_1, hub_2), binary. Whether a connection from hub *hub_1* to hub *hub_2* exists and allows a flow of *element*. Note that X_N(hub_1, hub_2) is different from X_N(hub_2, hub_1).
- Y_N(hub_1, hub_2), binary. Whether a connection between hubs *hub_1* and *hub_2* exists and allows a flow of *element*.
- X_SYS(hub), binary. Whether the hub *hub* is connected to at least one other hub, no matter the direction of the connection.
- For all t, F_SYS(hub, t), continuous, in kW. The power related to *element* going from hub *hub* to the network.
- For all t, F_N(hub_1, hub_2, t), continuous, in kW. The power related to *element* going from hub *hub_1* to hub *hub_2* through the network. Note that F_N(hub_1, hub_2, t) is the opposite of F_N(hub_2, hub_1, t).

NonThermalNetwork components declare the following KPIs:

- *COST_network* In euros. Contributes to the "Eco" objective function.

Parameters

- **hubs_locations** (*dict {Hub: (float, float)}*) –
  - Keys of *hubs_locations* are the hubs possibly connected by the network. They define the *hubs* attribute.
  - Values of *hubs_locations* define x and y coordinates in space. In km. They describe the position of the hub given the absolute reference (0, 0). Used to calculate distance between two hubs. The used distance function can be accessed by the *get_distance_function* function and set using *set_distance_function* from tamos.network.
- **element** (ElectricityVector, FuelVector) – Element exchanged between *hubs*.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the network. The *properties* attribute must include the following keys:
  - "Losses (%/km)"
  - "CAPEX (EUR/km)"
  - "OPEX (%CAPEX)"
- **production_hub** (Hub) – The hub that bears all the distribution losses of the network. Must be one of *hubs* and must be able to send *element* to the network.
- **scale_factor** (*float, optional, default 1*) – Multiplies the two coordinates of each hub in *hubs_locations*. A scale factor >1 tends to increase the distance between two hubs.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.
- **name** (*str, optional*) –

Notes

1. A network component describe an oriented graph where each node is a hub and each edge is a connection between two hubs.
2. Connection status between two hubs can be defined using the *set_connection_status* and *set_node_status* methods. Per default, all pairs of hubs are connected according to the status *no_connection*.
3. The connection of production hub to every other hub using the network is not mandatory, i.e. nothing constrains the network graph to be connected.
4. The way distribution losses are declared and applied to *production_hub* in the MILP formalism gives no upper bound for these losses. This implementation is chosen to prevent the use of the absolute value of a continuous variable, which comes with a binary variable. The consequence of this implementation is that problem where large losses benefit to the minimization of the objective function (unlikely to happen) might get a solution with no physical meaning.

Methods

__init__(hubs_locations, element, ...[, ...])	NonThermalNetwork instances makes possible to share ElectricityVector or FuelVector elements between hubs.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
generate_MSP([excluded_hubs])	Defines a minimum spanning tree (MSP) linking all hubs of *hubs* that are not in *excluded_hubs*.
get_connection_power_bounds(hub_1, hub_2)	Returns the power limits of the flow of element from hub *hub_1* to hub *hub_2*.
get_connection_status(hub_1, hub_2)	Returns the status of the directional connection between two hubs.
get_distance(hub_1, hub_2)	Returns the distance between two hubs.
might_connect(hub)	Checks whether a hub is able to connect to the network.
plot()	Plots a representation of the network hubs and connections in the (x, y) space.
set_connection_power_bounds(hub_1, hub_2[, ...])	Sets power limits on the flow of element from hub *hub_1* to hub *hub_2*.
set_connection_status(hub_1, hub_2, status)	Defines the connection status of the edge going from *hub_1* to *hub_2*.
set_node_status(hub, status)	Defines the connection status of every incoming and outcoming edge of a hub.
set_status(status[, excluded_hubs])	Define the connection status of all edges except the ones involving a hub from *excluded_hubs*.

Attributes

connection	
eco_count	Whether this instance contributes to the system "Eco" KPI bool
element	Element exchanged between *hubs*.
hubs	The hubs involved in the network definition.
name	str.
no_connection	
optim_one_way_max	
optim_one_way_min	
optim_two_ways	
production_hub	The hub that bears all the distribution losses of the network.
scale_factor	Multiplies the two coordinates of each hub in *hubs_locations*.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI bool

property element

Element exchanged between *hubs*.

Whether *element* is cooled down or warmed up does not define if the network is a heating or cooling network.

generate_MSP(*excluded_hubs=None*)

Defines a minimum spanning tree (MSP) linking all hubs of *hubs* that are not in *excluded_hubs*.

The MSP calculation does not account for bidirectionality thus all edges of the MSP graph are given the status *optim_two_ways*. The status of the other edges (not part of the MSP) are not modified.

Parameters

excluded_hubs (*list of Hub, optional*) – All hubs from *excluded_hubs* must be part of *hubs*.

get_connection_power_bounds(*hub_1*, *hub_2*)

Returns the power limits of the flow of element from hub *hub_1* to hub *hub_2*.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.

- **hub_2** (Hub) – Must be hubs from *hubs*.

Return type
The lower bound and upper bound of the power flow from *hub_1* to *hub_2*, in kW.

`get_connection_status`(*hub_1*, *hub_2*)

Returns the status of the directional connection between two hubs.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.

Returns
The status of the edge from hub_1 to hub_2.

Return type
network status

`get_distance`(*hub_1*, *hub_2*)

Returns the distance between two hubs. The used distance function can be accessed by the *get_distance_function* function and set using *set_distance_function* from tamos.network.

Parameters

- **hub_1** (Hub) – Must be from *hubs*.
- **hub_2** (Hub) – Must be from *hubs*.

Return type
The distance from *hub_1* to *hub_2* which is the same than from *hub_2* to *hub_1*

`property hubs`

The hubs involved in the network definition.

Some of these hubs might be completely disconnected from the network (i.e. network.might_connect(hub) is False) yet they are still considered and associated with MILP decision variables.

`might_connect`(*hub*)

Checks whether a hub is able to connect to the network.

Parameters
hub (Hub) –

Returns

- *True if the following two conditions are met*
- * Hub *hub* is one of *hubs*.
- * There exists at least one hub *hub_2* in *hubs* such that connection status from *hub* to *hub_2* or *hub_2* to *hub* – is different from 'No connection'.

`property name`

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

`plot()`

Plots a representation of the network hubs and connections in the (x, y) space.

A call to this method gives a visual insight of how the network is parametrized, BEFORE optimization. The optimization implicitely tranforms every edge status to either 'No connection' or 'Connection'.

Notes

The line linking two hubs is straight for commodity and does not represent the real distance function used in the MILP model.

`property production_hub`

The hub that bears all the distribution losses of the network. Must be one of *hubs* and must be able to send *element* to the network.

`property scale_factor`

Multiplies the two coordinates of each hub in *hubs_locations*.

A scale factor >1 tends to increase the distance between two hubs. float

`set_connection_power_bounds`(*hub_1*, *hub_2*, *power_lb=None*, *power_ub=None*)

Sets power limits on the flow of element from hub *hub_1* to hub *hub_2*. These limits apply only if the connection from *hub_1* to *hub_2* is used.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.
- **power_lb** (*int, float or numpy.ndarray*) – power_lb >= 0, power_ub >= 0 In kW. The lower bound (upper bound) of the flow of *element* from *hub_1* to *hub_2*.
- **power_ub** (*int, float or numpy.ndarray*) – power_lb >= 0, power_ub >= 0 In kW. The lower bound (upper bound) of the flow of *element* from *hub_1* to *hub_2*.

Examples

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=400, power_
↪ub=2000)
```

If the connection *hub_1* to *hub_2* exists, the power that flows from *hub_1* to *hub_2* must always be in the range [0.4, 2] MW.

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=1000)
>>> network.set_connection_power_bounds(hub_2, hub_1, power_ub=-1000)
```

Both calls perform the same operation, but second call is forbidden to make things clearer.

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=1000)
>>> network.set_connection_power_bounds(hub_2, hub_1, power_lb=2000)
```

The constraints implied by these calls make impossible the existence of both connections (from *hub_1* to *hub_2* and *hub_2* to *hub_1*): if 1000 kW of *element* flows from *hub_1* to *hub_2* then -1000 k> flows from *hub_2* to *hub_1* (and vice versa).

set_connection_status(*hub_1*, *hub_2*, *status*)

Defines the connection status of the edge going from *hub_1* to *hub_2*.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.
- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden
  - connection: flow of *element* is possible
  - optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

Notes

Given two hubs *hub_1* and *hub_2*, the order of calls to *set_connection_status* matters.

```
>>> network.set_connection_status(hub_1, hub_2, network.no_direction)
>>> network.set_connection_status(hub_2, hub_1, network.optim_one_way_max)
>>> network.get_connection_status(hub_2, hub_1) == network.get_connection_
↪status(hub_1, hub_2)
    True
```

The first line has no effect because the second one defines again the connection from *hub_1* to *hub_2*.

set_node_status(*hub*, *status*)

Defines the connection status of every incoming and outcoming edge of a hub.

Parameters

- **hub** (Hub) – Must be from *hubs*.
- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden
  - connection: flow of *element* is possible
  - optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

- optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

set_status(*status*, *excluded_hubs=None*)

Define the connection status of all edges except the ones involving a hub from *excluded_hubs*.

Parameters

- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden
  - connection: flow of *element* is possible
  - optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
- **excluded_hubs** (*list of Hub, optional*) – All hubs from *excluded_hubs* must be part of *hubs*.

property used_elements

Elements used by the component.

6.1.2 tamos.network.ThermalNetwork

class tamos.network.**ThermalNetwork**(*hubs_locations*, *element*, *properties*, *production_hub*, *production_mode='heat&cold'*, *scale_factor=1*, *eco_count=True*, *name=None*)

__init__(*hubs_locations*, *element*, *properties*, *production_hub*, *production_mode='heat&cold'*, *scale_factor=1*, *eco_count=True*, *name=None*)

ThermalNetwork instances makes possible to share ThermalVectorPair elements between hubs.

Power is exchanged between two hubs, given a distribution losses proportional to the difference between network temperature and soil temperature. All distribution losses must be compensated for by an additional power in *production_hub*. The investment cost is proportional to the network length.

NonThermalNetwork components are associated with the following exported decision variables:

- X_N(hub_1, hub_2), binary. Whether a connection from hub *hub_1* to hub *hub_2* exists and allows a flow of *element*. Note that X_N(hub_1, hub_2) is different from X_N(hub_2, hub_1).
- Y_N(hub_1, hub_2), binary. Whether a connection between hubs *hub_1* and *hub_2* exists and allows a flow of *element*.
- X_SYS(hub), binary. Whether the hub *hub* is connected to at least one other hub, no matter the direction of the connection.

- For all t, F_SYS(hub, t), continuous, in kW. The power related to *element* going from hub *hub* to the network.
- For all t, F_N(hub_1, hub_2, t), continuous, in kW. The power related to *element* going from hub *hub_1* to hub *hub_2* through the network. Note that F_N(hub_1, hub_2, t) is the opposite of F_N(hub_2, hub_1, t).

NonThermalNetwork components declare the following KPIs:

- *COST_network* In euros. Contributes to the "Eco" objective function.

Parameters

- **hubs_locations** (*dict {Hub: (float, float)}*) –
  - Keys of *hubs_locations* are the hubs possibly connected by the network. They define the *hubs* attribute.
  - Values of *hubs_locations* define x and y coordinates in space. In km. They describe the position of the hub given the absolute reference (0, 0). Used to calculate distance between two hubs. The used distance function can be accessed by the *get_distance_function* function and set using *set_distance_function* from tamos.network.
- **element** (*ThermalVectorPair*) – Element exchanged between *hubs*. Whether *element* is cooled down or warmed up does not define if the network is a heating or cooling network.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the network. The *properties* attribute must include the following keys:
  - "Losses (%/km)"
  - "CAPEX (EUR/km)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
- **production_hub** (Hub) – The hub that bears:
  - all the distribution losses of the network.
  - the costs associated with the "Variable OPEX (EUR/MWh)" property.

  Must be one of *hubs* and must be able to exchange *element* with the network.
- **production_mode** (*{"heat&cold", "heat", "cold"}, optional, default "heat&cold"*) – Related to *production_hub*. Used to speed up the KPI declaration regarding the *Variable OPEX (EUR/MWh)* property. Provides insight on the sign of the flow going from *production_hub* to the network. Whether *element* is cooled down or warmed up is independent from *production_mode*.
  - "heat" ("cold"): only heat (cold) is send to the network by *production_hub*. The flow is always of the same sign, thus the KPI constraint is like: energy = sum(a_given_sign * power(t) * dt)
  - "heat" ("cold"): only heat (cold) is send to the network by *production_hub*. The sign of the flow may change during the operation period, thus the KPI constraint is like: energy = sum(abs(power(t)) * dt)

  Specifying "heat" or "cold" will speed up the KPI declaration but describes a particular state of energy flows. Specifying "heat&cold" makes the KPI declaration long (but has no impact on resolution) but works in all cases.

- **scale_factor** (*float, optional, default 1*) – Multiplies the two coordinates of each hub in *hubs_locations*. A scale factor >1 tends to increase the distance between two hubs.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.
- **name** (*str, optional*) –

Notes

1. A network component describe an oriented graph where each node is a hub and each edge is a connection between two hubs.
2. Connection status between two hubs can be defined using the *set_connection_status* and *set_node_status* methods. Per default, all pairs of hubs are connected according to the status *no_connection*.
3. The connection of *production_hub* to every other hub using the network is not mandatory, i.e. nothing constrains the network graph to be connected.
4. The variable OPEX applies only on the thermal production of *production_hub*. Another MILP implementation could have taken into account all hubs sending power to the network, at the cost of additional continuous decision variables and constraints.

Methods

__init__(hubs_locations, element, ...[, ...])	ThermalNetwork instances makes possible to share ThermalVectorPair elements between hubs.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
generate_MSP([excluded_hubs])	Defines a minimum spanning tree (MSP) linking all hubs of *hubs* that are not in *excluded_hubs*.
get_connection_power_bounds(hub_1, hub_2)	Returns the power limits of the flow of element from hub *hub_1* to hub *hub_2*.
get_connection_status(hub_1, hub_2)	Returns the status of the directional connection between two hubs.
get_distance(hub_1, hub_2)	Returns the distance between two hubs.
might_connect(hub)	Checks whether a hub is able to connect to the network.
plot()	Plots a representation of the network hubs and connections in the (x, y) space.
set_connection_power_bounds(hub_1, hub_2[, ...])	Sets power limits on the flow of element from hub *hub_1* to hub *hub_2*.
set_connection_status(hub_1, hub_2, status)	Defines the connection status of the edge going from *hub_1* to *hub_2*.
set_node_status(hub, status)	Defines the connection status of every incoming and outcoming edge of a hub.
set_soil_properties(soil_temperature, U[, ...])	Sets the physical properties that define thermal losses.
set_status(status[, excluded_hubs])	Define the connection status of all edges except the ones involving a hub from *excluded_hubs*.

Attributes

connection	
eco_count	Whether this instance contributes to the system "Eco" KPI bool
element	Element exchanged between *hubs*.
hubs	The hubs involved in the network definition.
name	str.
no_connection	
optim_one_way_max	
optim_one_way_min	
optim_two_ways	
production_hub	The hub that bears:
production_mode	Related to *production_hub*.
scale_factor	Multiplies the two coordinates of each hub in *hubs_locations*.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI bool

property element

Element exchanged between *hubs*.

Whether *element* is cooled down or warmed up does not define if the network is a heating or cooling network.

generate_MSP(*excluded_hubs=None*)

Defines a minimum spanning tree (MSP) linking all hubs of *hubs* that are not in *excluded_hubs*.

The MSP calculation does not account for bidirectionality thus all edges of the MSP graph are given the status *optim_two_ways*. The status of the other edges (not part of the MSP) are not modified.

Parameters

excluded_hubs (*list of Hub, optional*) – All hubs from *excluded_hubs* must be part of *hubs*.

get_connection_power_bounds(*hub_1*, *hub_2*)

Returns the power limits of the flow of element from hub *hub_1* to hub *hub_2*.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.

Return type

The lower bound and upper bound of the power flow from *hub_1* to *hub_2*, in kW.

get_connection_status(*hub_1*, *hub_2*)

Returns the status of the directional connection between two hubs.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.

Returns

The status of the edge from hub_1 to hub_2.

Return type

network status

get_distance(*hub_1*, *hub_2*)

Returns the distance between two hubs. The used distance function can be accessed by the *get_distance_function* function and set using *set_distance_function* from tamos.network.

Parameters

- **hub_1** (Hub) – Must be from *hubs*.
- **hub_2** (Hub) – Must be from *hubs*.

Return type
The distance from *hub_1* to *hub_2* which is the same than from *hub_2* to *hub_1*

property hubs

The hubs involved in the network definition.

Some of these hubs might be completely disconnected from the network (i.e. network.might_connect(hub) is False) yet they are still considered and associated with MILP decision variables.

might_connect(*hub*)

Checks whether a hub is able to connect to the network.

Parameters
hub (Hub) –

Returns

- *True if the following two conditions are met*
- * Hub *hub* is one of *hubs*.
- * There exists at least one hub *hub_2* in *hubs* such that connection status from *hub* to *hub_2* or *hub_2* to *hub* – is different from 'No connection'.

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

plot()

Plots a representation of the network hubs and connections in the (x, y) space.

A call to this method gives a visual insight of how the network is parametrized, BEFORE optimization. The optimization implicitely tranforms every edge status to either 'No connection' or 'Connection'.

Notes

The line linking two hubs is straight for commodity and does not represent the real distance function used in the MILP model.

property production_hub

The hub that bears:

- all the distribution losses of the network.
- the costs associated with the "Variable OPEX (EUR/MWh)" property.

Must be one of *hubs* and must be able to exchange *element* with the network.

property production_mode

Related to *production_hub*.

{"heat&cold", "heat", "cold"}, optional, default "heat&cold" Used to speed up the KPI declaration regarding the *Variable OPEX (EUR/MWh)* property. Provides insight on the sign of the flow going from *production_hub* to the network. Whether *element* is cooled down or warmed up is independent from *production_mode*.

- "heat" ("cold"): only heat (cold) is send to the network by *production_hub*. The flow is always of the same sign, thus the KPI constraint is like: energy = sum(a_given_sign * power(t) * dt)

- "heat" ("cold"): only heat (cold) is send to the network by *production_hub*. The sign of the flow may change during the operation period, thus the KPI constraint is like: energy = sum(abs(power(t)) * dt)

Specifying "heat" or "cold" will speed up the KPI declaration but describes a particular state of energy flows. Specifying "heat&cold" makes the KPI declaration long (but has no impact on resolution) but works in all cases.

property scale_factor

Multiplies the two coordinates of each hub in *hubs_locations*.

A scale factor >1 tends to increase the distance between two hubs. float

set_connection_power_bounds(*hub_1*, *hub_2*, *power_lb=None*, *power_ub=None*)

Sets power limits on the flow of element from hub *hub_1* to hub *hub_2*. These limits apply only if the connection from *hub_1* to *hub_2* is used.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.
- **power_lb** (*int, float or numpy.ndarray*) – power_lb >= 0, power_ub >= 0 In kW. The lower bound (upper bound) of the flow of *element* from *hub_1* to *hub_2*.
- **power_ub** (*int, float or numpy.ndarray*) – power_lb >= 0, power_ub >= 0 In kW. The lower bound (upper bound) of the flow of *element* from *hub_1* to *hub_2*.

Examples

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=400, power_
↪ub=2000)
```

If the connection *hub_1* to *hub_2* exists, the power that flows from *hub_1* to *hub_2* must always be in the range [0.4, 2] MW.

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=1000)
>>> network.set_connection_power_bounds(hub_2, hub_1, power_ub=-1000)
```

Both calls perform the same operation, but second call is forbidden to make things clearer.

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=1000)
>>> network.set_connection_power_bounds(hub_2, hub_1, power_lb=2000)
```

The constraints implied by these calls make impossible the existence of both connections (from *hub_1* to *hub_2* and *hub_2* to *hub_1*): if 1000 kW of *element* flows from *hub_1* to *hub_2* then -1000 k> flows from *hub_2* to *hub_1* (and vice versa).

set_connection_status(*hub_1*, *hub_2*, *status*)

Defines the connection status of the edge going from *hub_1* to *hub_2*.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.
- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden

- connection: flow of *element* is possible
- optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
- optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
- optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

Notes

Given two hubs *hub_1* and *hub_2*, the order of calls to *set_connection_status* matters.

```
>>> network.set_connection_status(hub_1, hub_2, network.no_direction)
>>> network.set_connection_status(hub_2, hub_1, network.optim_one_way_max)
>>> network.get_connection_status(hub_2, hub_1) == network.get_connection_
↪status(hub_1, hub_2)
    True
```

The first line has no effect because the second one defines again the connection from *hub_1* to *hub_2*.

set_node_status(*hub*, *status*)

Defines the connection status of every incoming and outcoming edge of a hub.

Parameters

- **hub** (Hub) – Must be from *hubs*.
- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden
  - connection: flow of *element* is possible
  - optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

set_soil_properties(*soil_temperature*, *U*, *losses_direction: Both'*, *'Heat losses'*, *'Heat gains = 'Both'*)

Sets the physical properties that define thermal losses.

Parameters

- **soil_temperature** (*int, float or numpy.ndarray*) – In Kelvins (K). The temperature of the soil at the buried depth of the network pipes.

- **U** (*int, float or numpy.ndarray*) – In W/(m.K). Specific heat loss per routed meter. Includes both supply and return pipes.
- **losses_direction** (*{'Both', 'Heat losses', 'Heat gains'}, optional, default 'Both'*) – Direction of the heat exchanges between the network infrastructure and the soil. Specifying 'Heat losses' ('Heat gains') allows to prevent from happening the case where cold (heat) must be produced in *production_hub* to compensate thermal gains (losses) in a district heating (district cooling) network, when thermal demand is lower than soil thermal exchanges.
  - 'Heat losses': only thermal energy exchanges from the network to the soil are taken into account, others are set to 0.
  - 'Heat gains': only thermal energy exchanges from the soil to the network are taken into account, others are set to 0.
  - 'Both': all thermal energy exchanges are taken into account.

Notes

1. The thermal energy exchanged between the network infrastructure and the soil is like: thermal_exchanges(t) = sign * U * network_length * ((T_warm(t)+T_cold(t)) / 2 - T_soil(t)) With:
   - network_length: the length of all the edges in the network. If connections are bidirectional, length is accounted for only once.
   - T_warm(t): temperature of the warm vector of *element*
   - T_cold(t): temperature of the cold vector of *element*
   - T_soil(t): temperature of the soil
2. By default, set_soil_properties is called with *soil_temperature* =273+10, *U* =0.7, *losses_direction* ='Both'.
3. This method does not assume pipes are laid down underground: setting a *U* value according to the *soil_temperature* value is enough to describe any surrounding environment of the pipes.

set_status(*status*, *excluded_hubs=None*)

Define the connection status of all edges except the ones involving a hub from *excluded_hubs*.

Parameters

- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden
  - connection: flow of *element* is possible
  - optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

- **excluded_hubs** (*list of Hub, optional*) – All hubs from *excluded_hubs* must be part of *hubs*.

property used_elements

Elements used by the component.

6.1.3 tamos.network.HREThermalNetwork

class tamos.network.**HREThermalNetwork**(*hubs_locations*, *element*, *properties*, *production_hub*, *production_mode='heat&cold'*, *scale_factor=1*, *eco_count=True*, *name=None*)

__init__(*hubs_locations*, *element*, *properties*, *production_hub*, *production_mode='heat&cold'*, *scale_factor=1*, *eco_count=True*, *name=None*)

HREThermalNetwork is a thermal network model that differs from *ThermalNetwork* by an investment cost being dependant from the annual network linear energy density. It is an adaptation of the model described in (Persson et al., 2019) ([1],[2]).

Power is exchanged between two hubs, given a distribution losses proportional to the difference between network temperature and soil temperature. All distribution losses must be compensated for by an additional power in *production_hub*.

NonThermalNetwork components are associated with the following exported decision variables:

- X_N(hub_1, hub_2), binary. Whether a connection from hub *hub_1* to hub *hub_2* exists and allows a flow of *element*. Note that X_N(hub_1, hub_2) is different from X_N(hub_2, hub_1).
- Y_N(hub_1, hub_2), binary. Whether a connection between hubs *hub_1* and *hub_2* exists and allows a flow of *element*.
- X_SYS(hub), binary. Whether the hub *hub* is connected to at least one other hub, no matter the direction of the connection.
- For all t, F_SYS(hub, t), continuous, in kW. The power related to *element* going from hub *hub* to the network.
- For all t, F_N(hub_1, hub_2, t), continuous, in kW. The power related to *element* going from hub *hub_1* to hub *hub_2* through the network. Note that F_N(hub_1, hub_2, t) is the opposite of F_N(hub_2, hub_1, t).

NonThermalNetwork components declare the following KPIs:

- *COST_network* In euros. Contributes to the "Eco" objective function.

Parameters

- **hubs_locations** (*dict {Hub: (float, float)}*) –
  - Keys of *hubs_locations* are the hubs possibly connected by the network. They define the *hubs* attribute.
  - Values of *hubs_locations* define x and y coordinates in space. In km. They describe the position of the hub given the absolute reference (0, 0). Used to calculate distance between two hubs. The used distance function can be accessed

[1] Persson U, Wiechers E, Möller B, Werner S. Heat Roadmap Europe: Heat distribution costs. Energy 2019;176:604–22. https://doi.org/10.1016/j.energy.2019.03.189.

[2] Persson U, Werner S. Heat distribution and the future competitiveness of district heating. Applied Energy 2011;88:568–76. https://doi.org/10.1016/j.apenergy.2010.09.020.

by the *get_distance_function* function and set using *set_distance_function* from tamos.network.

- **element** (*ThermalVectorPair*) – Element exchanged between *hubs*. Whether *element* is cooled down or warmed up does not define if the network is a heating or cooling network.
- **properties** (*dict {str: int | float}*) – Techno-economic properties of the network. The *properties* attribute must include the following keys:
  - "Losses (%/km)"
  - "OPEX (%CAPEX)"
  - "Variable OPEX (EUR/MWh)"
  - "Used network length (m)"
  - "Capex subsidy (%)"
- **production_hub** (Hub) – The hub that bears:
  - all the distribution losses of the network.
  - the costs associated with the "Variable OPEX (EUR/MWh)" property.
  - the investment cost, which is associated with the properties "OPEX (%CAPEX)", "Used network length (m)" and "Capex subsidy (%)"

  Must be one of *hubs* and must be able to exchange *element* with the network.
- **production_mode** (*{"heat&cold", "heat", "cold"}, optional, default "heat&cold"*) – Related to *production_hub*. Used to speed up the KPI declaration regarding the *Variable OPEX (EUR/MWh)* property. Provides insight on the sign of the flow going from *production_hub* to the network. Whether *element* is cooled down or warmed up is independent from *production_mode*.
  - "heat" ("cold"): only heat (cold) is send to the network by *production_hub*. The flow is always of the same sign, thus the KPI constraint is like: energy = sum(a_given_sign * power(t) * dt)
  - "heat" ("cold"): only heat (cold) is send to the network by *production_hub*. The sign of the flow may change during the operation period, thus the KPI constraint is like: energy = sum(abs(power(t)) * dt)

  Specifying "heat" or "cold" will speed up the KPI declaration but describes a particular state of energy flows. Specifying "heat&cold" makes the KPI declaration long (but has no impact on resolution) but works in all cases.
- **scale_factor** (*float, optional, default 1*) – Multiplies the two coordinates of each hub in *hubs_locations*. A scale factor >1 tends to increase the distance between two hubs.
- **eco_count** (*bool, optional, default True*) – Whether this instance contributes to the system "Eco" KPI.
- **name** (*str, optional*) –

Notes

1. A network component describe an oriented graph where each node is a hub and each edge is a connection between two hubs.
2. Connection status between two hubs can be defined using the *set_connection_status* and *set_node_status* methods. Per default, all pairs of hubs are connected according to the status *no_connection*.
3. The connection of *production_hub* to every other hub using the network is not mandatory, i.e. nothing constrains the network graph to be connected.
4. The variable OPEX applies only on the thermal production of *production_hub*. Another MILP implementation could have taken into account all hubs sending power to the network, at the cost of additional continuous decision variables and constraints.
5. The linear heat density calculation implies a non-linear expression involving the network length L. This model relies on the network length being a parameter, "Used network length (m)".
6. The property "Capex subsidy (%)" decreases the investment cost but does not impact the variable cost.

References

Methods

__init__(hubs_locations, element, ...[, ...])	HREThermalNetwork is a thermal network model that differs from *ThermalNetwork* by an investment cost being dependant from the annual network linear energy density.
compute_actualized_cost(CAPEX, OPEX, ...[, ...])	Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.
generate_MSP([excluded_hubs])	Defines a minimum spanning tree (MSP) linking all hubs of *hubs* that are not in *excluded_hubs*.
get_connection_power_bounds(hub_1, hub_2)	Returns the power limits of the flow of element from hub *hub_1* to hub *hub_2*.
get_connection_status(hub_1, hub_2)	Returns the status of the directional connection between two hubs.
get_distance(hub_1, hub_2)	Returns the distance between two hubs.
might_connect(hub)	Checks whether a hub is able to connect to the network.
plot()	Plots a representation of the network hubs and connections in the (x, y) space.
set_connection_power_bounds(hub_1, hub_2[, ...])	Sets power limits on the flow of element from hub *hub_1* to hub *hub_2*.
set_connection_status(hub_1, hub_2, status)	Defines the connection status of the edge going from *hub_1* to *hub_2*.
set_node_status(hub, status)	Defines the connection status of every incoming and outcoming edge of a hub.
set_soil_properties(soil_temperature, U[, ...])	Sets the physical properties that define thermal losses.
set_status(status[, excluded_hubs])	Define the connection status of all edges except the ones involving a hub from *excluded_hubs*.

Attributes

connection	
eco_count	Whether this instance contributes to the system "Eco" KPI bool
element	Element exchanged between *hubs*.
hubs	The hubs involved in the network definition.
name	str.
no_connection	
optim_one_way_max	
optim_one_way_min	
optim_two_ways	
production_hub	The hub that bears:
production_mode	Related to *production_hub*.
scale_factor	Multiplies the two coordinates of each hub in *hubs_locations*.
used_elements	Elements used by the component.

compute_actualized_cost(*CAPEX*, *OPEX*, *system_lifetime*, *lifetime=None*, *discount_rate=None*)

Computes the cost of a component using its 'Lifetime' and 'Discount rate (%)' properties.

Parameters

- **CAPEX** (*float*) – Capital Expenditure. Cost in euros paid every *technical_lifetime* periods.
- **OPEX** (*float*) – Operational Expenditure. Cost in euros paid each period.
- **system_lifetime** (*int*) – Number of periods defining the existence of the energy system.
- **lifetime** (*int, optional*) – Number of periods defining the existence of the component. If specified, overwrite the "Lifetime" property.
- **discount_rate** (*float*) – In percent (%). Describes the importance of the economic amortization process, per period. If specified, overwrite the "Discount rate (%)" property.

Returns

- *A 3-tuple (total_cost, CAPEX_share, OPEX_share) where*
- * CAPEX_share is the share of total cost related to *CAPEX*
- * OPEX_share is the share of total cost related to *OPEX*
- ** total_cost = CAPEX_share + OPEX_share*

Notes

Takes into account residual value of component in the case *system_lifetime* is not a multiple of *lifetime*. In this case, the last replacement occuring at period replacement_period is paid in proportion of 'CAPEX' depending linearly on the number of periods left: CAPEX * (system_lifetime - replacement_period) / lifetime

property eco_count

Whether this instance contributes to the system "Eco" KPI bool

property element

Element exchanged between *hubs*.

Whether *element* is cooled down or warmed up does not define if the network is a heating or cooling network.

generate_MSP(*excluded_hubs=None*)

Defines a minimum spanning tree (MSP) linking all hubs of *hubs* that are not in *excluded_hubs*.

The MSP calculation does not account for bidirectionality thus all edges of the MSP graph are given the status *optim_two_ways*. The status of the other edges (not part of the MSP) are not modified.

Parameters

excluded_hubs (*list of Hub, optional*) – All hubs from *excluded_hubs* must be part of *hubs*.

get_connection_power_bounds(*hub_1*, *hub_2*)

Returns the power limits of the flow of element from hub *hub_1* to hub *hub_2*.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.

Return type

The lower bound and upper bound of the power flow from *hub_1* to *hub_2*, in kW.

get_connection_status(*hub_1*, *hub_2*)

Returns the status of the directional connection between two hubs.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.

Returns

The status of the edge from hub_1 to hub_2.

Return type

network status

get_distance(*hub_1*, *hub_2*)

Returns the distance between two hubs. The used distance function can be accessed by the *get_distance_function* function and set using *set_distance_function* from tamos.network.

Parameters

- **hub_1** (Hub) – Must be from *hubs*.
- **hub_2** (Hub) – Must be from *hubs*.

Return type
The distance from *hub_1* to *hub_2* which is the same than from *hub_2* to *hub_1*

property hubs

The hubs involved in the network definition.

Some of these hubs might be completely disconnected from the network (i.e. network.might_connect(hub) is False) yet they are still considered and associated with MILP decision variables.

might_connect(*hub*)

Checks whether a hub is able to connect to the network.

Parameters
hub (Hub) –

Returns

- *True if the following two conditions are met*
- * Hub *hub* is one of *hubs*.
- * There exists at least one hub *hub_2* in *hubs* such that connection status from *hub* to *hub_2* or *hub_2* to *hub* – is different from 'No connection'.

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ' ' { } ~.

Type
Name of the instance

plot()

Plots a representation of the network hubs and connections in the (x, y) space.

A call to this method gives a visual insight of how the network is parametrized, BEFORE optimization. The optimization implicitely tranforms every edge status to either 'No connection' or 'Connection'.

Notes

The line linking two hubs is straight for commodity and does not represent the real distance function used in the MILP model.

property production_hub

The hub that bears:

- all the distribution losses of the network.
- the costs associated with the "Variable OPEX (EUR/MWh)" property.

Must be one of *hubs* and must be able to exchange *element* with the network.

property production_mode

Related to *production_hub*.

{"heat&cold", "heat", "cold"}, optional, default "heat&cold" Used to speed up the KPI declaration regarding the *Variable OPEX (EUR/MWh)* property. Provides insight on the sign of the flow going from *production_hub* to the network. Whether *element* is cooled down or warmed up is independent from *production_mode*.

- "heat" ("cold"): only heat (cold) is send to the network by *production_hub*. The flow is always of the same sign, thus the KPI constraint is like: energy = sum(a_given_sign * power(t) * dt)

- "heat" ("cold"): only heat (cold) is send to the network by *production_hub*. The sign of the flow may change during the operation period, thus the KPI constraint is like: energy = sum(abs(power(t)) * dt)

Specifying "heat" or "cold" will speed up the KPI declaration but describes a particular state of energy flows. Specifying "heat&cold" makes the KPI declaration long (but has no impact on resolution) but works in all cases.

property scale_factor

Multiplies the two coordinates of each hub in *hubs_locations*.

A scale factor >1 tends to increase the distance between two hubs. float

set_connection_power_bounds(*hub_1*, *hub_2*, *power_lb=None*, *power_ub=None*)

Sets power limits on the flow of element from hub *hub_1* to hub *hub_2*. These limits apply only if the connection from *hub_1* to *hub_2* is used.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.
- **power_lb** (*int, float or numpy.ndarray*) – power_lb >= 0, power_ub >= 0 In kW. The lower bound (upper bound) of the flow of *element* from *hub_1* to *hub_2*.
- **power_ub** (*int, float or numpy.ndarray*) – power_lb >= 0, power_ub >= 0 In kW. The lower bound (upper bound) of the flow of *element* from *hub_1* to *hub_2*.

Examples

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=400, power_
↪ub=2000)
```

If the connection *hub_1* to *hub_2* exists, the power that flows from *hub_1* to *hub_2* must always be in the range [0.4, 2] MW.

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=1000)
>>> network.set_connection_power_bounds(hub_2, hub_1, power_ub=-1000)
```

Both calls perform the same operation, but second call is forbidden to make things clearer.

```
>>> network.set_connection_power_bounds(hub_1, hub_2, power_lb=1000)
>>> network.set_connection_power_bounds(hub_2, hub_1, power_lb=2000)
```

The constraints implied by these calls make impossible the existence of both connections (from *hub_1* to *hub_2* and *hub_2* to *hub_1*): if 1000 kW of *element* flows from *hub_1* to *hub_2* then -1000 k> flows from *hub_2* to *hub_1* (and vice versa).

set_connection_status(*hub_1*, *hub_2*, *status*)

Defines the connection status of the edge going from *hub_1* to *hub_2*.

Parameters

- **hub_1** (Hub) – Must be hubs from *hubs*.
- **hub_2** (Hub) – Must be hubs from *hubs*.
- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden

- connection: flow of *element* is possible
- optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
- optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
- optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

Notes

Given two hubs *hub_1* and *hub_2*, the order of calls to *set_connection_status* matters.

```
>>> network.set_connection_status(hub_1, hub_2, network.no_direction)
>>> network.set_connection_status(hub_2, hub_1, network.optim_one_way_max)
>>> network.get_connection_status(hub_2, hub_1) == network.get_connection_
↪status(hub_1, hub_2)
    True
```

The first line has no effect because the second one defines again the connection from *hub_1* to *hub_2*.

set_node_status(*hub*, *status*)

Defines the connection status of every incoming and outcoming edge of a hub.

Parameters

- **hub** (Hub) – Must be from *hubs*.
- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden
  - connection: flow of *element* is possible
  - optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

set_soil_properties(*soil_temperature*, *U*, *losses_direction: Both'*, *'Heat losses'*, *'Heat gains = 'Both'*)

Sets the physical properties that define thermal losses.

Parameters

- **soil_temperature** (*int, float or numpy.ndarray*) – In Kelvins (K). The temperature of the soil at the buried depth of the network pipes.

- **U** (*int, float or numpy.ndarray*) – In W/(m.K). Specific heat loss per routed meter. Includes both supply and return pipes.
- **losses_direction** (*{'Both', 'Heat losses', 'Heat gains'}, optional, default 'Both'*) – Direction of the heat exchanges between the network infrastructure and the soil. Specifying 'Heat losses' ('Heat gains') allows to prevent from happening the case where cold (heat) must be produced in *production_hub* to compensate thermal gains (losses) in a district heating (district cooling) network, when thermal demand is lower than soil thermal exchanges.
  - 'Heat losses': only thermal energy exchanges from the network to the soil are taken into account, others are set to 0.
  - 'Heat gains': only thermal energy exchanges from the soil to the network are taken into account, others are set to 0.
  - 'Both': all thermal energy exchanges are taken into account.

Notes

1. The thermal energy exchanged between the network infrastructure and the soil is like: thermal_exchanges(t) = sign * U * network_length * ((T_warm(t)+T_cold(t)) / 2 - T_soil(t)) With:
   - network_length: the length of all the edges in the network. If connections are bidirectional, length is accounted for only once.
   - T_warm(t): temperature of the warm vector of *element*
   - T_cold(t): temperature of the cold vector of *element*
   - T_soil(t): temperature of the soil
2. By default, set_soil_properties is called with *soil_temperature* =273+10, *U* =0.7, *losses_direction* ='Both'.
3. This method does not assume pipes are laid down underground: setting a *U* value according to the *soil_temperature* value is enough to describe any surrounding environment of the pipes.

set_status(*status*, *excluded_hubs=None*)

Define the connection status of all edges except the ones involving a hub from *excluded_hubs*.

Parameters

- **status** (*status* may take 5 values that are attributes of this instance:) –
  - no_connection: flow of *element* is forbidden
  - connection: flow of *element* is possible
  - optim_one_way_min: flow of *element* must exist in at least one direction (from hub_1 to hub_2, from hub_2 to hub_1 or in both directions) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_one_way_max: flow of *element* may exist in at most one direction (from hub_1 to hub_2 or from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.
  - optim_two_ways: flow of *element* may exist in both directions (from hub_1 to hub_2 and from hub_2 to hub_1) This status also defines the opposite status, e.g. defining 'hub_1 to hub_2' status defines the one of 'hub_2 to hub_1'.

- **excluded_hubs** (*list of Hub, optional*) – All hubs from *excluded_hubs* must be part of *hubs*.

property used_elements

Elements used by the component.

6.2 Utilities

tamos.network.get_distance_function()	Returns the function that gives the distance in km between two hubs.
tamos.network.set_distance_function(...)	Defines the function that gives the distance in km between two hubs.

6.2.1 tamos.network.get_distance_function

tamos.network.**get_distance_function**()

Returns the function that gives the distance in km between two hubs.

6.2.2 tamos.network.set_distance_function

tamos.network.**set_distance_function**(*distance_function*)

Defines the function that gives the distance in km between two hubs.

Parameters

- **distance_function** (*callable, f(u, v), that meets the following definition:*) –
- **[x** (** u is a list*) –
- **v).** (*y] where x and y are space coordinates of point u (same for*) –
- **f(u** (**) –
- **f(v** (*v) =*) –
- **u)** –
- **f(u** –
- **km** (*v) returns a distance in*) –

Notes

Default distance function is *scipy.spatial.distance.cityblock*.

CHAPTER

SEVEN

CONSTRAINING COMPONENTS AND THE USE OF COMPONENTS

tamos.InterfaceMask(component[, element, ...])	

tamos.Hub.components_assemblies	Components assemblies of the hub.
tamos.MILPModel.components_assemblies	Components assemblies of the model.

7.1 tamos.InterfaceMask

class tamos.**InterfaceMask**(*component*, *element=None*, *name=None*, *power_lb=None*, *power_ub=None*, *energy_lb=None*, *energy_ub=None*)

__init__(*component*, *element=None*, *name=None*, *power_lb=None*, *power_ub=None*, *energy_lb=None*, *energy_ub=None*)

Defines power and energy constraints regarding exchanges between a component and the hub it is affected to.

Must be passed to a hub to be effective. An InterfaceMask instance can be affected to different hubs. Constraints are applied in an independent manner no matter the hubs it is affected to.

Parameters

- **component** (*production, storage, element_IO or network instance*) –
- **element** (*element instance, optional*) – Must be provided only if *component* is a production or storage instance.
- **name** (*str, optional*) –
- **power_lb** (*numpy.ndarray or float, optional*) – The lower bound (upper bound) of the power related to the element *element* (if relevant) of component *component*.
- **power_ub** (*numpy.ndarray or float, optional*) – The lower bound (upper bound) of the power related to the element *element* (if relevant) of component *component*.
- **energy_lb** (*float, optional*) – The lower bound (upper bound) of the energy related to the element *element* (if relevant) of component *component*. Energy is calculated as sum(power(t) * dt(t)) on the entire operation period.
- **energy_ub** (*float, optional*) – The lower bound (upper bound) of the energy related to the element *element* (if relevant) of component *component*. Energy is calculated as sum(power(t) * dt(t)) on the entire operation period.

Examples

```
>>> tms.InterfaceMask(component=electric_heater, element=electricity, energy_
↪ub=1000)
```

The amount of energy related to element *electricity* during the entire operation period must not exceed 1000 kWh.

```
>>> tms.InterfaceMask(component=electric_heater, element=heat, power_lb=5)
```

The amount of energy related to element *heat* must be always greater than 5 kW.

```
>>> tms.InterfaceMask(component=electric_heater, element=heat, power_lb=numpy.
↪linspace(1, 10, 8760),
...                                                    power_ub=numpy.
↪linspace(1, 10, 8760))
```

The amount of energy related to element *heat* must increase linearly with time, from 1 to 10 kW.

Methods

__init__(component[, element, name, ...])	Defines power and energy constraints regarding exchanges between a component and the hub it is affected to.

Attributes

component	The component whose element exchanges with the hub interface are constrained.
element	If *component* is a storage or production instance, the element whose power flows and energy balances are constrained.
energy_lb	The lower bound of the energy related to the element *element* (if relevant) of component *component*.
energy_ub	The upper bound of the energy related to the element *element* (if relevant) of component *component*.
name	str.
power_lb	The lower bound of the power related to the element *element* (if relevant) of component *component*.
power_ub	The upper bound of the power related to the element *element* (if relevant) of component *component*.

property component

The component whose element exchanges with the hub interface are constrained.

property element

If *component* is a storage or production instance, the element whose power flows and energy balances are constrained.

property energy_lb

The lower bound of the energy related to the element *element* (if relevant) of component *component*. Energy is calculated as sum(power(t) * dt(t)) on the entire optimization period. None or float.

property energy_ub

The upper bound of the energy related to the element *element* (if relevant) of component *component*. Energy is calculated as sum(power(t) * dt(t)) on the entire optimization period. None or float.

property name

str. This name is used in MILP model description. names must not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: ! " # $ % & , . ; ? @ _ ‘ ’ { } ~.

Type
Name of the instance

property power_lb

The lower bound of the power related to the element *element* (if relevant) of component *component*. None, numpy.ndarray or float.

property power_ub

The upper bound of the power related to the element *element* (if relevant) of component *component*. None, numpy.ndarray or float.

7.2 tamos.Hub.components_assemblies

property Hub.**components_assemblies**

Components assemblies of the hub.

Must be provided as a list of 3-tuple objects (n_min, n_max, components) where:

- *n_min* (*n_max*) is the minimum (maximum) number of components from *components* that must be installed in the hub.
- *components* is a component or list of production, storage or element_IO components. If one component of *components* is not one of hub components, the 3-tuple (n_min, n_max, components) is ignored during constraints declaration.

Examples

```
>>> hub.components_assemblies = [(1, 2, [heat_load_1, heat_load_2]), (1, 1, heat_
↪load_3), (0, 1, [electric_heater_1, electric_heater_2])])
```

At least one of the first and second loads must be used. The third load must be used. At most one of the two heaters must be used.

```
>>> hub.components_assemblies = [(0, 1, [electric_heater, heat_grid]), (1, 1, heat_
↪load)])
```

The load must be used. Either electric_heater or heat_grid can be used.

7.3 tamos.MILPModel.components_assemblies

property MILPModel.**components_assemblies**

Components assemblies of the model.

Must be provided as a list of 3-tuple objects (n_min, n_max, components) where:

- *n_min* (*n_max*) is the minimum (maximum) number of components from *components* that must be installed, all hubs of *hubs* included.
- *components* is a component or list of production, storage or element_IO components. For any component of *components*, if this component is not in at least one hub of this MILPModel instance, the 3-tuple (n_min, n_max, components) is ignored during constraints declaration.

Examples

```
>>> hub_1 = Hub(components=[heat_load_1, heat_load_2])
>>> hub_2 = Hub(components=[heat_load_1])
>>> hub_3 = Hub(components=[heat_load_2])
>>> MILPModel = MILPModel(hubs=[hub_1, hub_2, hub_3])
>>> MILPModel.components_assemblies = [(0, 1, heat_load_1), (2, 3, [heat_load_1,␣
↪heat_load_2])]
```

heat_load_1 might be used at most one time, all hubs [hub_1, hub_2, hub_3] included. the number of times heat_load_1 or heat_load_2 are used in all hubs [hub_1, hub_2, hub_3] is greater than 2 but smaller than 3.

CHAPTER

EIGHT

DATA INPUT AND OUTPUT

8.1 Export and aggregate results

tamos.data_IO.ResultsExport(MILPModel[, ...])	
tamos.data_IO.ResultsBatch(working_dir[, name])	

8.1.1 tamos.data_IO.ResultsExport

class tamos.data_IO.**ResultsExport**(*MILPModel*, *get_LP=False*, *get_MPS=False*, *parent_working_dir=None*, *csv_precision=3*, *replace_inverted_TVP=False*)

__init__(*MILPModel*, *get_LP=False*, *get_MPS=False*, *parent_working_dir=None*, *csv_precision=3*, *replace_inverted_TVP=False*)

ResultsExport instances extract and write to disk the decision variables and KPIs of a solved MILPModel instance.

Variable and KPIs values are collected at the instance creation. These are stored as attributes. A call to a *write_* method copies this information on disk as CSV and text files. A call to *dump_object* method write a binary form of this instance on disk.

Parameters

- **MILPModel** (*MILPModel*) – If the solving of the MILPModel instance did not succeed, only pre solve model information is collected by ResultsExport. If the solving succeeded, KPIs, decision variables and some non-essential model results are collected.
- **get_LP** (*bool, optional, default, False*) – Whether to collect the model following the LP standard.
- **get_MPS** (*bool, optional, default False*) – Whether to collect the model following the MPS standard.
- **parent_working_dir** (*str or path-like, optional*) – This defines the *working_dir* attribute according to working_dir=parent_working_dir/name, where 'name' is the *name* attribute of MILPModel. The export of results is done in *working_dir*. If not provided, *parent_working_dir* is a new directory called 'results' in the parent working directory.
- **csv_precision** (*int, optional, default 3*) – Number of decimals regarding CSV export of decision variables and KPIs.

- **replace_inverted_TVP** (*bool, optional, False*) –
  - If True, ThermalVectorPair elements '~TVP' are replaced by 'TVP' if both of them are used in the model. This does not change the values of decision variables and KPIs since a flow associated with 'TVP' is the opposite of the same flow associated with '~TVP'. Setting `replace_inverted_TVP`=True can make the results reading easier.
  - If False, both '~TVP' and 'TVP' are kept.

Notes

A ResultsExport instance is a representation of a MILPModel instance at the time it is created. The further change in the MILPModel instance are not reflected in the already existing ResultsExport instance.

Methods

__init__(MILPModel[, get_LP, get_MPS, ...])	ResultsExport instances extract and write to disk the decision variables and KPIs of a solved MILPModel instance.
dump_object()	Writes the binary form of this instance.
load_object(path)	Loads a binary ResultsExport or ResultsBatch object.
reduce_memory(df)	Reduces the memory used by a DataFrame by changing the data type of numerical columns.
remove_small_values(df, name[, threshold, ...])	Removes small numerical values from a dataframe containing decision variable or KPI values.
write_KPIs()	Writes all KPIs defined at component level in a directory named 'KPIs'.
write_LP()	Writes the model according to Cplex LP standard.
write_MPS()	Writes the model according to the MPS standard.
write_all()	Writes all the content of *MILPModel* collected by this instance.
write_description()	Writes the *description* attribute of *MILPModel*.
write_exec_time()	Writes the time spent on:
write_model_complexity()	Writes the number of decision variables and constraints per type (continuous, discrete, etc...).
write_solution_summary()	Writes important information about the solved model, including the values of the 3 main KPIs, "Eco", "CO2" and "Exergy".
write_unsatisfied_constraints()	Writes every constraint of the model that is unsatisfied after model solve, given a tolerance of 1e-6.
write_variables()	Writes all decision variables in a directory named 'Variables'.

Attributes

working_dir	Directory the results are read from or exported to.

dump_object()

Writes the binary form of this instance.

Notes

1. Model data (decision variables, constraints, KPIs, etc…) are lost during this operation.
2. The binary form of this instance is called 'results *name*' with *name* the *name* attribute of *MILPModel*.
3. The *description* attribute of *MILPModel* is also dumped to speed up the call of *ResultsBatch.create_batch* method.

static load_object(*path*)

Loads a binary ResultsExport or ResultsBatch object.

Parameters
path (*str or path-like*) –

Return type
The loaded object.

static reduce_memory(*df*)

Reduces the memory used by a DataFrame by changing the data type of numerical columns.

Data type of float and integer columns is changed for the type with the smallest memory usage.

Parameters
df (*DataFrame*) –

Notes

This method works inplace, i.e. *df* is modified.

static remove_small_values(*df*, *name*, *threshold=0.1*, *drop_zeros=None*)

Removes small numerical values from a dataframe containing decision variable or KPI values.

Parameters

- **df** (*pandas.DataFrame*) –
- **name** (*str*) – Name of the column of *df* that contains the numerical values.
- **threshold** (*int or float, optional, default 0.1*) – All values 'x' for which abs(x) < *threshold* are replaced by 0.
- **drop_zeros** (*{'Series', 'All'}, optional*) –
  - If not provided, all rows where *name* value is zero are kept.
  - 'Series': each temporal data series is kept if at least one value in this serie is not zero. *df* must include a 'Date' column for *drop_zeros* `=='Series' to be effective. *A temporal data serie of* `df` is a subset of df such that only the 'Date' and *name* columns have changing values in this subset.
  - 'All' : remove all rows where *name* value is zero.

Notes

This method works inplace, i.e. *df* is modified. Near-zero values in column *name* are removed according to *threshold* and *drop_zeros*.

property working_dir

Directory the results are read from or exported to.

write_KPIs()

Writes all KPIs defined at component level in a directory named 'KPIs'.

write_LP()

Writes the model according to Cplex LP standard.

The *get_LP* argument must have been set to 'True' at instance creation.

Notes

Cplex LP standard slightly differs from the conventional LP standard. Prefer the use *write_MPS* in case a communication with another solver is needed

write_MPS()

Writes the model according to the MPS standard.

The *get_MPS* argument must have been set to 'True' at instance creation.

write_all()

Writes all the content of *MILPModel* collected by this instance.

write_description()

Writes the *description* attribute of *MILPModel*.

write_exec_time()

Writes the time spent on:

- decision variables declaration ('MILPModel.declare_variables')
- constraints and KPIs declaration ('MILPModel.declare_constraints_and_KPIs')
- solving process, in seconds ('MILPModel.solve')
- solving process, in deterministic Cplex ticks ('Solving deterministic time')
- total time (variables, constraints, KPIs, solving), in seconds ('Total time')

write_model_complexity()

Writes the number of decision variables and constraints per type (continuous, discrete, etc…).

write_solution_summary()

Writes important information about the solved model, including the values of the 3 main KPIs, "Eco", "CO2" and "Exergy".

write_unsatisfied_constraints()

Writes every constraint of the model that is unsatisfied after model solve, given a tolerance of 1e-6.

write_variables()

Writes all decision variables in a directory named 'Variables'.

8.1.2 tamos.data_IO.ResultsBatch

class tamos.data_IO.**ResultsBatch**(*working_dir*, *name=None*)

__init__(*working_dir*, *name=None*)

Do not use. See *create_batch* and *create_batch_from_binaries* class methods.

Methods

__init__(working_dir[, name])	Do not use.
create_batch(working_dir, description_matcher)	A ResultsBatch instance binds ResultsExport instances together to ease the process of results analysis.
create_batch_from_binaries(working_dir, ...)	A ResultsBatch instance binds ResultsExport instances together to ease the process of results analysis.
dump_object()	Writes the binary form of this instance in the directory *working_dir*.
load_object(path)	Loads a binary ResultsExport or ResultsBatch object.
reduce_memory(df)	Reduces the memory used by a DataFrame by changing the data type of numerical columns.
remove_descriptors(descriptors[, check_unique])	Removes some descriptors, i.e. keys of attribute *relevant_descriptor*.
remove_small_values(df, name[, threshold, ...])	Removes small numerical values from a dataframe containing decision variable or KPI values.
sum(RBs)	Defines a unique ResultsBatch instance that is the sum of several instances.
write_all()	Writes the *vars_*, *KPIs* and *results* attributes as CSV files, in the directory *working_dir*.

Attributes

working_dir	Directory the results are read from or exported to.

classmethod create_batch(*working_dir*, *description_matcher*, *keep_unknown_file_descriptor=True*, *keep_unknown_matcher_descriptor=True*, *upfront_processing_func=None*, *keep_results_exports=False*, *name=None*, ***upfront_processing_kwargs*)

A ResultsBatch instance binds ResultsExport instances together to ease the process of results analysis.

Important attributes of ResultsBatch instances are: * *vars_* : concatenation of the decision variables of the ResultsExport instances * *KPIs* : concatenation of the KPIs of the ResultsExport instances * *results* : other information about ResultsExport instances

This method creates a ResultsBatch instance from ResultsExports stored on disk. It must be used when various optimization results are stored in the same directory and clearly identified by the *description* attribute of MILPModel instances.

The ResultsExport instances to keep are specified using the *keep_unknown_file_descriptor* and *keep_unknown_matcher_descriptor* attributes.

Parameters

- **working_dir** (*str or path-like*) – The directory containing the ResultsExport results. This directory is typically the same than the one specified for the *parent_working_dir* argument of ResultsExport instances. This directory is also the one where the aggregated results will be written.
- **description_matcher** (*dict {str: list(int | float | str) | None}*) – Any ResultsExport in *working_dir* having a *description* attribute matching *description_matcher* is kept. Let *key* and *values* such that description_matcher[key] = values.
  - If *values* is None, every ResultsExports having *key* as a key of its *description* attribute is kept.
  - Else, *values* is a list of int, float and str. A ResultsExports having *key* as a key of its *description* attribute is kept only if description[key] is one of *values*.
- **keep_unknown_file_descriptor** (*bool, optional, default True*) – If False, ResultsExports instances whose *description* attribute contains keys that are not keys of *description_matcher* are discarded. If True, these instances are kept only if the the conditions specified by the *description_matcher* attribute are met.
- **keep_unknown_matcher_descriptor** (*bool, optional, default True*) – If False, ResultsExports instances whose *description* attribute lacks keys that are keys of *description_matcher* are discarded. If True, these instances are kept only if the the conditions specified by the *description_matcher* attribute are met.
- **upfront_processing_func** (*callable f(x), optional.*) – Useful to process ResultsExport instances before they are aggregated into one single ResultsBatch. f must accept a ResultsExport instance and performs inplace. Attribute *description* of *x* must not be modified.
- **keep_results_exports** (*bool, optional, default False*) – If True, all the ResultsExport instances that match the selection criteria are kept in the *results_exports* attribute (dict).
- **name** (*str, optional*) – Name of this instance. Setting a name may prevent the deletion of existing files during disk dump of this instance.
- **upfront_processing_kwargs** (*optional*) – Keyword arguments passed to the *upfront_processing_func* callable, like: f(x, **kwargs).

Returns
RB

Return type
ResultsBatch

Notes

1. The *relevant_descriptors* attribute is a dict summing-up the *description* attribute of the MILPModel instances.
2. The 'batch_analysis' Jupyter Notebook requires a ResultsBatch instance.

classmethod create_batch_from_binaries(*working_dir*, *results_exports: List of pp*, *upfront_processing_func=None*, *name=None*, ***upfront_processing_kwargs*)

A ResultsBatch instance binds ResultsExport instances together to ease the process of results analysis.

Important attributes of ResultsBatch instances are: * *vars_*: concatenation of the decision variables of the ResultsExport instances * *KPIs*: concatenation of the KPIs of the ResultsExport instances * *results*: other information about ResultsExport instances

This method creates a ResultsBatch instance from a list of ResultsExports.

Parameters

- **working_dir** (*str or path-like*) – The directory where the aggregated results will be written.
- **results_exports** (*list of ResultsExport*) – ResultsExport instances that define the ResultsBatch instance. Duplicates will be ignored.
- **upfront_processing_func** (*callable f(x), optional.*) – Useful to process ResultsExport instances before they are aggregated into one single ResultsBatch. f must accept a ResultsExport instance and performs inplace. Attribute *description* of *x* must not be modified.
- **name** (*str, optional*) – Name of this instance. Setting a name may prevent the deletion of existing files during disk dump of this instance.
- **upfront_processing_kwargs** (*optional*) – Keyword arguments passed to the *upfront_processing_func* callable, like: f(x, **kwargs).

Returns
RB

Return type
ResultsBatch

dump_object()

Writes the binary form of this instance in the directory *working_dir*.

static load_object(*path*)

Loads a binary ResultsExport or ResultsBatch object.

Parameters
path (*str or path-like*) –

Return type
The loaded object.

static reduce_memory(*df*)

Reduces the memory used by a DataFrame by changing the data type of numerical columns.

Data type of float and integer columns is changed for the type with the smallest memory usage.

Parameters
df (*DataFrame*) –

Notes

This method works inplace, i.e. *df* is modified.

remove_descriptors(*descriptors*, *check_unique=False*)

Removes some descriptors, i.e. keys of attribute *relevant_descriptor*.

Parameters

- **descriptors** (*list of str*) – Must be a subset of the keys of *relevant_descriptors*.
- **check_unique** (*bool, optional, default False*) – Describes the method behavior when any of the descriptor in *descriptors* is associated with multiple values:
  - If True, an AssertionError is raised.
  - If False, descriptor is removed, leading to possible duplicated rows in attributes *vars_*, *KPIs* and *results*

static remove_small_values(*df*, *name*, *threshold=0.1*, *drop_zeros=None*)

Removes small numerical values from a dataframe containing decision variable or KPI values.

Parameters

- **df** (*pandas.DataFrame*) –
- **name** (*str*) – Name of the column of *df* that contains the numerical values.
- **threshold** (*int or float, optional, default 0.1*) – All values 'x' for which abs(x) < *threshold* are replaced by 0.
- **drop_zeros** (*{'Series', 'All'}, optional*) –
  - If not provided, all rows where *name* value is zero are kept.
  - 'Series': each temporal data series is kept if at least one value in this serie is not zero. *df* must include a 'Date' column for *drop_zeros`='Series' to be effective. A temporal data serie of `df* is a subset of df such that only the 'Date' and *name* columns have changing values in this subset.
  - 'All' : remove all rows where *name* value is zero.

Notes

This method works inplace, i.e. *df* is modified. Near-zero values in column *name* are removed according to *threshold* and *drop_zeros*.

classmethod sum(*RBs*)

Defines a unique ResultsBatch instance that is the sum of several instances.

Attributes *vars_*, *KPIs* and *results* are concatenated and merged. Attributes *relevant_indicators* are merged.

Parameters
RBs (*list of ResultsBatch*) –

Returns
RB – A ResultsBatch instance

Return type
ResultsBatch

Notes

1. The addition operator '+' may be used instead of a call to *sum(RBs)* for *RBs* of length 2, i.e. these two lines perform the same operation:

```
>>> ResultsBatch.sum([RB1, RB2])
>>> RB1 + RB2
```

2. The *working_dir* attribute of *RB* is the one of the first element of *RBs*.

property working_dir

Directory the results are read from or exported to.

write_all()

Writes the *vars_*, *KPIs* and *results* attributes as CSV files, in the directory *working_dir*.

8.2 Load data from disk

tamos.data_IO.read_properties(path)	Read a CSV file describing techno-economic properties of a component.
tamos.data_IO.read_data(path, col_name[, ...])	Read a CSV file describing a serie of values

8.2.1 tamos.data_IO.read_properties

tamos.data_IO.**read_properties**(*path*)

Read a CSV file describing techno-economic properties of a component.

Parameters
path (*str or path-like object*) – The path of the file to load properties from.

Returns
properties – A dict of the content of the file located at *path*.

Return type
dict

Notes

1. The file must have two columns:
   - The first column contains properties name
   - The second column contains properties value
2. The file must not have a header row (i.e. title row).
3. Lines starting with '#' are treated as comments.

8.2.2 tamos.data_IO.read_data

tamos.data_IO.**read_data**(*path*, *col_name*, *start=None*, *end=None*)

Read a CSV file describing a serie of values

Parameters

- **path** (*str or path-like object*) – The path of the file to load values from.
- **col_name** (*str, optional*) – The name of the column to be loaded from the CSV file.
- **start** (*int, optional*) – First row of values being loaded. If not provided, start=0.
- **end** (*int, optional*) – Last row of values being loaded. If not provided, end is the last row of the file.

Return type

A numpy.ndarray of the content located at *path* under *col_name*.

Notes

1. The file must not have a header row (i.e. title row).
2. Lines starting with '#' are treated as comments.

CHAPTER

NINE

SOLVING TOOLS

tamos.solve_tools.AdvSolve(MILPModel, ...[, ...])	

9.1 tamos.solve_tools.AdvSolve

class tamos.solve_tools.**AdvSolve**(*MILPModel*, *working_dir*, *skip_existing=True*, ***kwargs*)

__init__(*MILPModel*, *working_dir*, *skip_existing=True*, ***kwargs*)

AdvSolve is a wrapper that ease the advanced solving of MILP models.

Attribute *last_binaries* contains the ResultsExport instances of of the latest solved models. These all are results of *MILPModel* yet following different solving configurations.

Parameters

- **MILPModel** (MILPModel) – A *MILPModel* instance whose *declare_variables* and *declare_constraints_and_KPIs* methods have already been called. *MILPModel* is used in every call to this instance methods.
- **working_dir** (*str or path-like*) – Directory the results are read from (see *skip_existing*) and written to.
- **skip_existing** (*bool, optional, True*) –
  - If True and if there exists a ResultsExport on disk having the same name than *MILPModel*, existing results are loaded instead of solving again *MILPModel*.
  - If False, *MILPModel* is solved in all cases.
- **kwargs** (*optional*) – Keywords arguments. Some are passed to the *solve* method of *MILPModel* instance. These are:
  - MIP_gap: float, default 1e-4
  - threads: int, default 0
  - timelimit: int, default 43200

  Others are passed to *ResultsExport*. These are:
  - get_LP: bool, default False
  - get_MPS: bool, default False
  - csv_precision: int, default 3

– replace_inverted_TVP: bool, default False

parent_working_dir of ResultsExport will be ignored since *working_dir* has the same use.

Notes

1. Each MILP solving is performed using the Cplex optimizer.
2. AdvSolve does not perform model declaration.

Methods

__init__(MILPModel, working_dir[, skip_existing])	AdvSolve is a wrapper that ease the advanced solving of MILP models.
`solve`(*args, **kwargs)	
`solve_front`(*args, **kwargs)	

Attributes

MILPModel	Used in every call to this instance methods.
kwargs	Keywords arguments.
last_binaries	The ResultsExport instances of the latest solved models.
skip_existing	• If True and if there exists a ResultsExport on disk having the same name than *MILPModel*,
working_dir	Directory the results are read from (see *skip_existing*) and written to.

property MILPModel

Used in every call to this instance methods.

property kwargs

Keywords arguments. Some are passed to the *solve* method of *MILPModel* instance. These are:

- MIP_gap: float, default 1e-4
- threads: int, default 0
- timelimit: int, default 43200

Others are passed to *ResultsExport*. These are:

- get_LP: bool, default False
- get_MPS: bool, default False
- csv_precision: int, default 3
- replace_inverted_TVP: bool, default False

parent_working_dir of ResultsExport will be ignored since *working_dir* has the same use.

property last_binaries

The ResultsExport instances of the latest solved models. dict {str: ResultsExport}

property skip_existing

- If True and if there exists a ResultsExport on disk having the same name than *MILPModel*, existing results are loaded instead of solving again *MILPModel*.
- If False, *MILPModel* is solved in all cases.

property working_dir

Directory the results are read from (see *skip_existing*) and written to.

CHAPTER

TEN

GENERAL SETTINGS

tamos.allow_duplicated_names(allow)	Defines the policy regarding components sharing the same name.
tamos.use_name_in_MILP(bool)	Defines whether the name of components is used to give an exhaustive human-friendly name to decision variables, constraints and KPIs in the MILP problem.
tamos.reset_names_list()	Forgets every component name.

10.1 tamos.allow_duplicated_names

tamos.**allow_duplicated_names**(*allow*)

Defines the policy regarding components sharing the same name.

Parameters

allow (*bool*) –

- If True, duplicated names might exist. Leads to problems in model export regarding the identification of different MILP objects.
- If False, names are changed during the component declaration so that no duplicate exists.

10.2 tamos.use_name_in_MILP

tamos.**use_name_in_MILP**(*bool*)

Defines whether the name of components is used to give an exhaustive human-friendly name to decision variables, constraints and KPIs in the MILP problem.

This has no effect on the exported variables and KPIs which will always be named according to the component name. Consider calling *use_name_in_MILP(True)* only for debugging and exporting LP or MPS files of the model with a human-friendly content.

Parameters

bool (*bool*) –

- If False (default), variables, constraints and KPIs will receive non human-friendly names.
- If True, components names are used. This has two consequences:
  - It slightlys lengthen the declaration time of the MILP problem.
  - It makes LP and MPS files more readable but with a bigger size on disk.

10.3 tamos.reset_names_list

tamos.**reset_names_list**()

Forgets every component name.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search