oemof heat documentation

Release 0.0.1

oemof developer group

Oct 28, 2020

Getting started

CHAPTER 1

Getting started

DHNx is a toolbox for optimization and simulation of district heating and cooling systems.

Contents

1.1 Using DHNx

1.1.1 Installation

If you have a working Python3 environment, use pypi to install the latest oemof version:

```
pip install dhnx
```

For Installing the latest (dev) version, clone DHNx from github:

```
git clone https://github.com/oemof/DHNx.git
```

Now you can install it your local version of DHNx using pip:

```
pip install -e <path/to/DHNx/root/dir>
```

1.1.2 Examples

Examples are provided here. Also, have a look at the *Examples* section for some more explanation.

1.2 Contributing to DHNx

Contributions are welcome. You can write issues to announce bugs or errors or to propose enhancements. Or you can contribute a new approach that helps to model district heating/cooling systems. If you want to contribute, fork the project at github, develop your features on a new branch and finally open a pull request to merge your contribution to DHNx.

As DHNx is part of the oemof developer group we use the same developer rules, described here.

CHAPTER 2

Examples

2.1 Create a thermal network

```
from dhnx.network import ThermalNetwork

thermal_network = ThermalNetwork()

thermal_network.add('Producer', id=0, lat=50, lon=10)

thermal_network.add('Consumer', id=0, lat=50, lon=10)

thermal_network.add('Pipe', id=0, from_node='producer-0', to_node='consumer-0')

print(thermal_network)

# returns
# dhnx.network.ThermalNetwork object with these components
#   * 1 producers
#   * 1 consumers
#   * 1 pipes

print(thermal_network.components.pipes)

# returns
#         from_node     to_node
#    0  producer-0  consumer-0
```

CHAPTER 3

Thermal Network

The thermal network is the central object in DHNx. It provides a container class that holds a number of components. All available components are defined in components.csv, which is rendered in the following table.

component_class	list_name	description
Environment	environment	Environment
Producer	producers	Heat producer
Consumer	consumers	Heat consumer
Fork	forks	Node where several pipes meet
Pipe	pipes	Pipes representing double pipes (feed and return) that connect nodes

Every component has a number of attributes which are defined in components_attrs/. Each attribute is given a name, type (`int`, `float`, `str` etc.), unit, default value, a description, a status (`Input` or `Output`) and requirement (`required` or `optional`).

The attributes are presented in detail in the following sections.

3.1 Consumer

Consumers are the nodes where the heat provided by the district heating network is actually used. They are characterized by these attributes:

attribute	type	unit	default	description	status	requirement
id	int	n/a	n/a	Unique id	Input	required
component_type	str	–	Consumer	Component type	Input	optional
lat	float	n/a	n/a	Geographic latitude	Input	optional
lon	float	n/a	n/a	Geographic longitude	Input	optional
mass_flow	float/sequence	kg/s	n/a	Mass flow	Input	optional
delta_temp_drop	float/sequence	kg/s	n/a	Temperature drop from inlet to return	Input	optional
zeta_inlet	float	–	n/a	Localized pressure loss coefficient for inlet flow	Input	optional
zeta_return	float	–	n/a	Localized pressure loss coefficient for return flow	Input	optional

3.2 Producer

A producer is a general node that provides heat to the district heating network. Producers are described with the following attributes:

attribute	type	unit	default	description	status	requirement
id	int	n/a	n/a	Unique id	Input	required
component_type	str	–	Producer	Component type	Input	optional
lat	float	n/a	n/a	Geographic latitude	Input	optional
lon	float	n/a	n/a	Geographic longitude	Input	optional
temp_inlet	float/sequence	deg C or K	n/a	Inlet temperature at producer	Input	optional
zeta_inlet	float	–	n/a	Localized pressure loss coefficient for inlet flow	Input	optional
zeta_return	float	–	n/a	Localized pressure loss coefficient for return flow	Input	optional

3.3 Fork

Forks are the nodes where several pipes of the network meet. Forks have the attributes described in the following table:

attribute	type	unit	default	description	status	requirement
id	int	n/a	n/a	Unique id	Input	required
component_type	str	–	Fork	Component type	Input	optional
lat	float	n/a	n/a	Geographic latitude	Input	optional
lon	float	n/a	n/a	Geographic longitude	Input	optional
zeta_inlet	float	–	n/a	Localized pressure loss coefficient for inlet flow	Input	optional
zeta_return	float	–	n/a	Localized pressure loss coefficient for return flow	Input	optional

3.4 Pipe

Pipes imply the feed and return pipes connecting the different nodes of the network. They are characterized by these attributes:

attribute	type	unit	default	description	status	requirement
id	int	n/a	n/a	Unique id	Input	required
component_type	str	–	Pipe	Component type	Input	optional
from_node	int	n/a	n/a	Node where Pipe begins	Input	required
to_node	int	n/a	n/a	Node where Pipe ends	Input	required
length	float	m	n/a	Length of the Pipe	Input	optional
diameter	float	mm	n/a	Inner diameter of the pipes	Input	optional
heat_transfer_coeff	float	W/(m*K)	n/a	Heat transfer coefficient	Input	optional
roughness	float	mm	n/a	Roughness of pipes	Input	optional

CHAPTER 4

Optimisation models

Generally, this library should allow the optimisation of district heating grids with various configurations settings and different approaches. The optimisation methods of this library are tools to assist the planning process of DHS projects and to analyze the economic feasibility of DHS for a given district, community or city - either by focusing on the DHS itself, or by also considering the overall energy system of a district, which could not just be the heating sector, but also the electricity, mobility sector or the gas infrastructure.

At the moment, there is one approach using *oemof-solph* as linear optimisation library implemented. This approach is explained in the following sections. It totally makes sense to have some experiences with *oemof-solph* to understand this toolbox more easily.

4.1 Scope

The following questions can be addressed using the *optimize_investment* method of the *ThermalNetwork*:

- What is the cost-optimal topology and dimensioning of a DHS piping system, given the locations of potential central heat supply plants, the potential locations for the DHS piping system (e.g. street network), and the position of consumers?
- In addition to the first question, what is the cost-optimal expansion of a given DHS system?
- Is it cost-efficient to build a DHS at all, if there a consumer-wise heat supply alternatives? (Comparison of central and de-central supply strategies)
- What is the optimal dispatch of the heat producers? (In case there are no expansion options, but just existing DHS pipes)
- Planned: Streets-wise aggregation option

To answer these questions, at the moment, the LP and MILP optimisation library oemof.solph is used. Other approaches, e.g. heuristic approaches, might follow.

The following sections will give an overview about the general usage/workflow, (the necessary input data, the different optimisation settings and options, the results), and second, the underlying mathematical description.

4.2 Usage

Links to the subsections:

- *Overview*
- *Input Data*
- *Label systematic*
- *Results*

4.2.1 Overview

The optimisation of a given *ThermalNetwork* is executed by:

```
import dhnx

tnw = dhnx.network.ThermalNetwork()

tnw = network.from_csv_folder('path/to/thermal_network')

invest_opt = dhnx.input_output.load_invest_options('path/to/invest_options')

tnw.optimize_investment(invest_options=invest_opt)
```

For executing an optimisation, you must provide *investment options* additional to the previous data, which defines a *ThermalNetwork*. Both are explained in the following section.

4.2.2 Input Data

In this section, it is firstly revised, what input data is exactly necessary from the *ThemalNetwork* class, and then explained, what data needs to be provided as *investment options*, and what *optimisation settings* you can apply.

The following figure provides an overview of the input data:

Fig. 1: Fig. 1: Optimisation Input Data

The structure of the input data might look a bit confusing at the beginning, but provides a lot of options for building up complex district heating models. There are two groups of data: Firstly, data that describes the components and the connectivity of the network, required by the ThermalNetwork class. Secondly, data that is necessary for the investment optimization. For now, all data needs to be provided in csv files. This means that you do not need to provide a geo-reference for applying an district heating network optimisation model at all. Probably, in many cases, it is the export of four geo-referenced layers (e.g. geopandasdataframe, shp-file, or any other), which are a line layer representing the potential places for the DHS-trenches, and three point layers for the producers, the consumers, and the potential forks of the DHS system. All geometry information of the network system is passed by an *id* for each element. Thus, the line layer connects all points and provides the spatial relation with the attributes *from_node*, *to_node*, and *length*. If you prepare the data, be careful that every consumer is connected to an pipe, and every piping network system is connected to at least one producer.

ThermalNetwork

The data for the *ThermalNetwork* must be provided in the structure as defined for the .csv reader. The following data is required for applying an optimisation:

```
tree
├── pipes.csv                       # (required)
├── consumers.csv                   # (required)
├── forks.csv                       # (required)
├── producers.csv                   # (required)
└── sequences                       # (optional)
    └── consumers-heat_flow.csv
```

The attributes, which are required, and which are optional with respect to the optimisation, are presented in detail in the following:

Pipes

The basis for the district heating system optimisation is a table of potential pipes. The following attributes of the *ThermalNetwork* must be given:

- **id**: see *Thermal Network*
- **from_node**: see *Thermal Network*
- **to_node**: see *Thermal Network*
- **length**: see *Thermal Network*

The following attributes are additional attributes of the optimisation module. These attributes are optional for the optimisation:

attribute	type	unit	default	description	status	requirement
existing	bool	n/a	0	Binary indicating and existing pipe	Input	optional
capacity	float	kW	0	Capacity for existing pipe	Input	optional
hp_type	object	n/a	'nan'	Type_label of existing pipe	Input	optional
active	bool	n/a	1	Binary indicating that edge is available	Input	optional
add_fix_costs	float	Eur/m	0	Additional fix investment costs	Input	optional

- **existing**: Binary indicating an existing pipe. If there is no column *existing* given, all Pipes are free for optimisation.
- **capacity**: Capacity of existing pipes. If *existing* is *True*, a *capacity* must be given.
- **hp_type**: Label of the type of pipe. The *hp_type* refers to a set of parameters of a pipeline component. The parameters for the *hp_type* must be given in the following table (see *network/pipes.csv*). If *existing* is *True*, a *hp_type* must be given.
- **active**: Binary indicating that this pipe is considered. If no column *active* is given, all pipe-options are active. With this attribute, single pipes can be switched on and off. This can be very useful, if different scenarios should be analyzed, e.g. you might like to make a given street/pipes unavailable.

Consumers

The following attributes of the *ThermalNetwork* must be given:

- **id**: see *Thermal Network*

The following attributes are additional attributes of the optimisation module, and optional:

attribute	type	unit	default	description	status	requirement
active	bool	n/a	1	Binary indicating that consumer is active	Input	optional
P_heat_max	float	kW	n/a	Maximum heat load of consumer	Input	optional

- **active**: Binary indicating that consumer-xy is considered. If no column *active* is given, all consumers are active. With this attribute, single consumers can be switched on and off (e.g. for scenario analysis with different connection quotes).
- **P_heat_max**: Maximum heat load of consumer. If no column *P_heat_max* is given, the maximum heat load is calculated from the heat demand series (see *consumers-heat_flow.csv*). Depending on the optimisation setting, *P_heat_max* or the demand series is used for the optimisation (see *Optimisation settings* for further information).

Producers

The following attributes of the *ThermalNetwork* must be given:

- **id**: see *Thermal Network*

The following attributes are additional attributes of the optimisation module, and optional:

attribute	type	unit	default	description	status	requirement
active	bool	n/a	1	Binary indicating that producer is active	Input	optional

- **active**: Binary indicating that producer is active. If no column *active* is given, all producers are active. With this attribute, single producers can be switched on and off (e.g. for scenario analysis for different supply plant positions.

Forks

The following attributes of the *ThermalNetwork* must be given:

- **id**: see *Thermal Network*

For Forks, no additional required or optional attributes are needed by the optimisation module.

Consumers-heat_flow

Providing consumers heat flow time series is optional, **but** either the consumers demand must be given in form of *P_heat_max* as attribute of the *consumers*, or in form of a heat_flow time series with the minimum length of 1.

The following table shows an example of a *consumers-heat_flow*:

timestep	0	1
0	8	12
1	10	10
2	9	7

The column index must be the consumers *id* (And be careful that the *dtype* also matches the *id* of the consumers!).

Investment and additional options

If you want to do an investment or an simple unit commitment optimisation using the *optimize_investment()* method of the *ThermalNetwork*, you need to provide some additional data providing the investment parameter. The following sheme illustrates the structure of the investment input data:

```
tree
├── network
|   └── pipes.csv           # (required)
|
├── consumers
|   ├── bus.csv             # (required)
|   ├── demand.csv          # (required)
|   ├── source.csv          # (optional)
|   ├── storages.csv        # (optional)
|   └── transformer.csv     # (optional)
|
└── producers
    ├── bus.csv             # (required)
    ├── demand.csv          # (optional)
    ├── source.csv          # (required)
    ├── storages.csv        # (optional)
    └── transformer.csv     # (optional)
```

The investment input data provides mainly all remaining parameters of the oemof solph components, which are not specific for a single pipe, producer or consumer.

The minimum of required data is a specification of the pipe parameters (costs, and losses), a (heat) bus and a heat demand at the consumers, and a (heat) bus and a heat source at the producers. The detailed attributes are described in the following sections.

network/pipes.csv

You need to provide data on the investment options for the piping system. The following table shows the minimal required data you need to provide:

label_3	active	nonconvex	l_factor	l_factor_fix	cap_max	cap_min	capex_pipes	fix_costs
pipe-typ-A	1	0	0	0	100000	0	0.5	0

Each row represents an investment option. Note this investment option creates an oemof-solph *Heatpipeline* component for each active pipe. The units are given es examples. There are no units implemented, everybody needs to care about consistent units in his own model. At the same time, everybody is free to choose his own units (energy, mass flow, etc.).

- **label_3**: Label of the third tag. See *Label system*.
- **active**: (0/1). If *active* is 0, this heatpipeline component is not considered. This attribute helps for easy selecting and deselecting different investment options.
- **nonconvex**: (0/1). Choose whether a convex or a nonconvex investment should be performed. This leads to a different meaning of the minimum heat transport capacity (*cap_min*). See *P_heat_max* is given, the maximum heat load is calculated from the heat demand series (see *consumers-heat_flow.csv*). Depending on the optimisation setting, *P_heat_max* or the demand series is used for the optimisation (see oemof-solph documentation for further information).

- **l_factor**: Relative thermal loss per length unit (e.g. [kW_loss/(m*kW_installed)]. Defines the loss factor depending on the installed heat transport capacity of the pipe. The *l_factor* is multiplied by the invested capacity in investment case, and by the given *capacity* for a specific pipe in case of existing DHS pipes.
- **l_factor_fix**: Absolute thermal loss per length unit (e.g. [kW/m]). In case of *nonconvex* is 1, the *l_factor_fix* is zero if no investement in a specific pipe element is done. Be careful, if *nonconvex* is 0, this creates a fixed thermal loss.
- **cap_max**: Maximum installable capacity (e.g. [kW]).
- **cap_min**: Minimum installable capacity (e.g. [kW]). Note that there is a difference if a *nonconvex* investment is applied (see oemof-solph documentation for further information).
- **capex_pipes**: Variable investment costs depending on the installed heat transport capacity (e.g. [€/kW]).
- **fix_costs**: Fix investment costs independent of the installed capacity (e.g. [€])

See the *Heatpipeline* API for further details about the attributes.

consumers/.

All data for initialising *oemof-solph* components at the consumers are provided by the .csv files of the consumers folder. For a principal understanding, check out the excel reader example of *oemof-solph*, which works the same way: oemof-solph excel reader example.

The minimum requirement for doing an DHS optimisation is to provide an demand at the consumers. Therefore, you need the following two .csv files: *bus.csv* specifies the *oemof-solph Bus* components, and *demand.csv* defines the *oemof.solph.Sink*.

Table 1: Example for table of *Buses*

label_2	active	excess	shortage	shortage costs	excess costs
heat	1	0	0	99999	99999

You must provide at least one bus, which has a label (*label_2*, see *Label system*), and needs to be *active*. Optionally, you can add an *excess* or a *shortage* with *shortage costs* or *excess costs* respectively. This might help to get an feasible optimisation problem, in case your solver says, 'infeasible', for finding the error.

Table 2: demand.csv

label_2	active	nominal_value
heat	1	1

The demand also needs to have a label (*label_2*, see *Label system*), has the option for deactivating certain demands by using the attribute *active*, and needs to have a specification for the *nominal_value*. The *nominal_value* scales your demand.

producers/.

The producers look quite similar as the consumers. The consumers are taking energy from the DHS system. That means, the energy need to be supplied somewhere, which makes some kind of source necessary. To connect a source in the oemof logic, there needs to be a *oemof.solph.Bus* to which the source is connected. The two files *bus.csv* and *source.csv* need to be provided:

Table 3: Example for table of *Buses*

label_2	active	excess	shortage	shortage costs	excess costs
heat	1	0	0	99999	99999

The *bus.csv* table works analog to the consumers (see *consumers/.*).

Table 4: source.csv

label_2	active
heat	1

You need to provide at least one source at the *source.csv* table. Additionally, there are already a couple of options for adding additional attributes of the *oemof.solph.FLow* to the source, e.g. *variable_costs*, *fix* feed-in series, and *min* and *max* restrictions.

Generally, with this structure at every producer and consumer multiple oemof components, like *transformer* and *storages* can be already added.

Optimisation settings

The following table shows all options for the optimisation settings (See also `setup_optimise_investment()`):

attribute	type	default	description
heat_demand	str	'scalar'	'scalar' or 'series'. 'scalar': Peak heat load. 'series': time-series is used as heat demand.
simultaneity	float	1	Simultaneity or concurrency factor
num_ts	int	1	Number of time steps of optimisation
time_res	float	1	Time resolution
start_date	str	'1/1/2018'	Startdate for oemof optimisation
frequence	str	'H'	Lenght of period
solver	str	'cbc'	Name of solver
solve_kw	dict	{'tee': True}	Solver kwargs
bidirectional_pipes	bool	*False*	Bidirectional pipes leads to bi-directional flow attributes at the heatpipeline components {'min': -1, bidirectional: True}
dump_path	str	None	If a dump path is provided, the oemof dump file is stored.
dump_name	str	dump.oemof	Name of dump file
print_logging_info	bool	*False*	There are still some helpful print statements.
write_lp_file	bool	*False*	Option of writing lp-file. The lp-file is stored in 'User/.oemof/lp_files/DHNx.lp'

Some more explanation:

- **heat_demand**: If you set *heat_demand* to *'scalar'*, *num_ts* is automatically 1, and the peak heat load is used as heat demand for the consumers. If you want to use a time series as heat demand, apply *'series'*.

4.2.3 Label systematic

In order to access the oemof-solph optimisation results, a label systematic containing a tuple with 4 items is used. Please check the basic example of oemof-solph for using tuple as label (oemof-solph example tuple as label).

The following table illustrates the systematic:

Table 5: Labelling system (bold: obligatory; italic: examples)

tag1: general classification	tag2: commodity	tag3: specification / oemof object	tag4: Specific id
consumers	**heat**	**source**	*forks-34*
producers	*electricity*	**demand**	*consumers-15*
infrastructure	*gas*	excess	*prdocuers-4*
	hydrogen	shortage	*forks-14-forks-27*
		pipe-typ-A	*forks-24-consumers-122*
		storage_xy	
		boiler_typ_xy	

The labels are partly given automatically by the oemof-solph model builder:

- **tag1: general classification**: This tag is given automatically depending on the spatial belonging. *Tag1* can be either *consumers* (consumer point layer), *producers* (producer point layer) or *infrastructure* (pipes and forks layer). See *Thermal Network*.
- **tag2: commodity**: This tag specifies the commodity, e.g. all buses and transformer (heatpipelines) of the DHS pipeline system have automatically the *heat* as *tag2*. For a transformer of the consumers or the producers the *tag2* is *None*, because a transformer usually connects two commodities, e.g. gas –> heat.
- **tag3: specification / oemof object**: The third tag indicates either the oemof object and is generated automatically (this is the case for *demand.csv*, *source.csv* and *bus.csv*), or is the specific *label_3* of the *pipes.csv*, *transformer.csv* or *storages.csv*.
- **tag4: id**: The last tag shows the specific spatial position and is generated automatically.

4.2.4 Results

For checking and analysing the results you can either select to write the investment results of the heatpipeline components in the Thermalnetwork. You will find the results there:

```
# pipe-specific investment results
results = network.results.optimization['components']['pipes']
```

The following tables provides an overview of the results table:

attribute	type	unit	description	status
id	object	n/a	Unique id (see *pipes of network*)	Input
from_node	object	n/a	Node where Edge begins (see *pipes of network*)	Input
to_node	object	n/a	Node where Edge ends (see *pipes of network*)	Input
length	float	m	Length of pipe (see *pipes of network*)	Input
hp_type	object	n/a	Label of pipe which got selected from *network/pipes.csv*	Result
capacity	float	kW	Installed pipe capacity	Result
direction	float	-1/0/1	Flow direction of pipe: 1 if direction corresponds to the *from_node/to_node* notation. -1: opposite direction. 0: no investment. This works only if the setting option *bidirectional_pipes* is set *False*.	Result
costs	float	Eur	Total cost of pipe element.	Result
losses	float	kW	Total losses of pipe element.	Result

You can also check out the detailed results of the oemof model, which are stored at:

```
# oemof-solph results "main"
r_oemof_main = network.results.optimization['oemof']

# oemof-solph results "meta"
r_oemof_meta = network.results.optimization['oemof_meta']
```

Or you can also dump the oemof results and analyze the results as described in oemof-solph handling results. The labelling systematic will help you to easily get want you want, check *Label system*.

4.3 Introducing example

The following sections illustrates some features of the DHNx investment optimisation library.

You can execute and reproduce the example with all figures, check the *introduction_example*.

```
import matplotlib.pyplot as plt
import dhnx


# Initialize thermal network
network = dhnx.network.ThermalNetwork()
network = network.from_csv_folder('twn_data')

# Load investment parameter
invest_opt = dhnx.input_output.load_invest_options('invest_data')

# plot network
static_map = dhnx.plotting.StaticMap(network)
```

(continues on next page)

(continued from previous page)

```
static_map.draw(background_map=False)
plt.title('Given network')
plt.scatter(network.components.consumers['lon'], network.components.consumers['lat'],
            color='tab:green', label='consumers', zorder=2.5, s=50)
plt.scatter(network.components.producers['lon'], network.components.producers['lat'],
            color='tab:red', label='producers', zorder=2.5, s=50)
plt.scatter(network.components.forks['lon'], network.components.forks['lat'],
            color='tab:grey', label='forks', zorder=2.5, s=50)
plt.text(-2, 32, 'P0', fontsize=14)
plt.text(82, 0, 'P1', fontsize=14)
plt.legend()
plt.show()
```

The following figure shows the initial status of an (thermal) network, which is examined in the following sections:

Fig. 2: Fig. 2: Introduction example

The network of Fig. 2 consists of two options for the heat *producers* ("P0" and "P1"), eight *consumers*, and 11 *forks*. Before running the whole script, we will have a brief look at some input data. Let's start with the *consumers.csv* (*"twn_data/consumers.csv"*):

Table 6: consumers.csv

id	lat	lon	P_heat_max
0	30	40	15
1	10	40	18
2	10	60	25
3	30	70	36
4	50	60	25
5	90	40	12
6	60	10	50
7	60	30	20

A peak heating load *P_heat_max* is given for every consumer within the thermal network input data (see *Thermal Network Input*). The heat load needs to be pre-calculated, or assumed. The geographical attributes *lat* and *lon* are optional, but needed for plotting purpose. The next table shows the input data of the heat pipeline elements (*"invest_data/network/pipes.csv"*):

Table 7: pipes.csv

label_3	active	nonconvex	l_factor	l_factor_fix	cap_max	cap_min	capex_pipes	fix_costs
pipe-typ-A	1	0	0.00001	0	100000	0	2	0

In the simplest (and most approximate) case, a linear correlation between the thermal capacity and the investment costs can be used. In this example, we assume costs of 2 € per kilowatt installed thermal capacity and meter trench length. As maximum capacity *cap_max*, we take a very high value to make sure that the total heat load of all consumers (including losses) can be supplied. Additionally, we assume a heat loss of 0.00001 kW/m. The parameters of the district heating pipes need to be pre-calculated depending on the piping system and technical data sheet of the manufacturer. (In future, some pre-calculation function might be added.) The length of each pipe, the costs and the losses are related to, must be given in the *pipes.csv* table of the *Thermal Network Input*). Next, we optimise the network and get the results:

```
network.optimize_investment(invest_options=invest_opt)

# get results
results_pipes = network.results.optimization['components']['pipes']
print(results_pipes[['from_node', 'to_node', 'hp_type', 'capacity', 'heat_loss[kW]',
                     'invest_costs[€]']])
```

Since we do not have any other costs than investment costs, we can check if our results have been correctly processed by comparing the objective of the optimisation problem with the sum of the investment costs of the single pipes, which should be the same:

```
# sum of the investment costs of all pipes
print(results_pipes[['invest_costs[€]']].sum())

# objective value of optimisation problem
print(network.results.optimization['oemof_meta']['objective'])
```

Next, we can transfer the results to a *ThermalNetwork*, which contains only the pipes with an investment (to avoid possible numerical inaccuracy, the criterion is > 0.001):

```
# assign new ThermalNetwork with invested pipes
twn_results = network
twn_results.components['pipes'] = results_pipes[results_pipes['capacity'] > 0.001]
```

Now, lets have a look at the optimisation results, and plot the pipes:

```
# plot invested pipes
static_map_2 = dhnx.plotting.StaticMap(twn_results)
static_map_2.draw(background_map=False)
plt.title('Given network')
plt.scatter(network.components.consumers['lon'], network.components.consumers['lat'],
            color='tab:green', label='consumers', zorder=2.5, s=50)
plt.scatter(network.components.producers['lon'], network.components.producers['lat'],
            color='tab:red', label='producers', zorder=2.5, s=50)
plt.scatter(network.components.forks['lon'], network.components.forks['lat'],
            color='tab:grey', label='forks', zorder=2.5, s=50)
plt.text(-2, 32, 'P0', fontsize=14)
plt.text(82, 0, 'P1', fontsize=14)
plt.legend()
plt.show()
```

. . . which should give:

Fig. 3: Fig. 3: Pipes with investment

The next thing is to deactivate one heat producer by setting the attribute *active* of producer *P1* to 0 (compare *Thermal Network Input*):

Table 8: producers.csv

id	lat	lon	active
0	30	0	1
1	0	80	0

Now, the plot of pipes with a positive investment should look like this:

There are many other options already implemented. For example:

Fig. 4: Fig. 4: Pipes with investment (only *P0*)

- Using time series as heat demand
- Doing redundancy analysis by setting *min* and *max* attributes to the producers' sources
- Adding other *oemof-solph* objects like *Transformer*, *Storages*, further *Buses*, *Sinks* and *Sources* to each producer and consumer
- Using discrete pipe data by using the *nonconvex* investment options

Have fun!

CHAPTER 5

Simulation models

For a more detailed representation of temperature effects and pressure losses in the district heating network, linear optimization models do not suffice. In this situation, a simulation model can be the right choice.

5.1 Scope

The following questions can be addressed using a simulation model:

- How do the heat losses in the network depend on the temperatures of inlet and return pipes and ambient temperature?
- How much energy is necessary for the pumps to overcome the pressure losses in the network?
- How do these properties behave if the supply temperatures change?

To answer these questions, data has to be present or assumptions have to be made about the pipe's physical properties and the temperature drop at the consumers. Have a look at the *overview table* to learn about all the variables and parameters involved.

Conversely, if these are not known, running an optimization model would be the better choice. It is also possible to couple the two approaches, running an optimization first and then investigating the detailed physical behaviour. To learn about this option, please refer to the section *model coupling*.

Currently, the available simulation model does not handle transient states (i.e. propagation of temperature fronts through the pipes). The model evaluates a steady state of the hydraulic and thermal physical equations. This also means that consecutive time steps are modelled independently and the behaviour of thermal storages cannot be represented. A dynamic simulation model may be implemented at a later point in time.

5.2 Usage

To use DHNx for a simulation, you need to provide input data in a defined form. The basic requirements are the same for all `ThermalNetwork` s, but some input data is specific to the simulation.

```
tree
├── consumers.csv
├── pipes.csv
├── forks.csv
├── producers.csv
└── sequences
    ├── consumers-mass_flow.csv
    ├── consumers-delta_temp_drop.csv
    ├── environment-temp_env.csv
    └── producers-temp_inlet.csv
```

To run a simulation, create a `ThermalNetwork` from the input data and simulate:

```
import dhnx

thermal_network = dhnx.network.ThermalNetwork('path-to-input-data')

thermal_network.simulate()
```

Figure 1 shows a sketch of a simple district heating network that illustrates how the variables that are determined in a simulation model run are attributed to different parts of a network. Pipes have the attributes mass flows, heat losses and pressure losses (distributed and localized). Temperatures of inlet and return flow are attributed to the different nodes. Pump power belongs to the producers which are assumed to include the pumps. Variables that describe the network as a whole are global heat losses and global pressure losses.

Fig. 1: Fig. 1: Schematic of a simple district heating network and the relevant variables for simulation.

The above-mentioned variables can be found in the results of a simulation, which come in the following structure:

```
results
├── global-heat_losses.csv
├── global-pressure_losses.csv
├── nodes-temp_inlet.csv
├── nodes-temp_return.csv
├── pipes-dist_pressure_losses.csv
├── pipes-heat_losses.csv
├── pipes_loc_pressure_losses.csv
├── pipes-mass_flow.csv
└── producers-pump_power.csv
```

5.3 Underlying Concept

Name	Math. symbol	Unit	Common values
Variables			
mass flow	$\dot{m}$	kg/s	
mean flow velocity	v	m/s	$1 - 3\,m/s$
pressure	p	$Pa \mid bar$	Nominal pressures PN16 or PN25
pressure difference	Δp	$Pa \mid bar$	max. $300\,Pa/m$
pump power	$P_{el,pump}$	W	
temperature	T	$°C$	Inlet pipe: $70 - 130°C$, Return pipe: $50 - 70°C$
ambient temperature	T_{env}	$°C$	$10°C$
heat flow	$\dot{Q}$	W	
Water properties			
density	ρ	kg/m^3	$971.78\,kg/m^3$ at $80°C$
spec. heat capacity	c	$J/(kgK)$	$4190\,J/(kgK)$ at $80°C$ and PN16 or PN25
dynamic viscosity	μ	$kg/(m \cdot s)$	$0.00035\,kg/(m \cdot s)$ at $80°C$ and PN16 or PN25
darcy friction factor	λ	–	
Reynolds number	Re	–	
Parameters			
pipe's length	L	m	
pipe's inner diameter	D	mm	Nominal diameters DN25 - DN250
localized pressure loss coefficient	ζ	–	$\zeta_{tee,split} = 2$, $\zeta_{tee,join} = 0.75$
standard acceleration due to gravity	g	m/s^2	$9.81\,m/s^2$
altitude difference	Δh	m	
pipe's absolute surface roughness	ϵ	mm	$0.01\,mm$
heat transfer coefficient	U (sometimes k)	$W/(Km^2)$	
spec. heat loss per meter	U_{spec} (sometimes just U)	$W/(Km)$	$0.15 - 0.9\,W/(Km)$
pump efficiency	η_{pump}	–	0.7
electric pump efficiency	η_{el}	–	
hydraulic pump efficiency	η_{hyd}	–	

The following equations are related to a single pipe.

5.3.1 Hydraulic equations

A pressure difference between two ends of a pipe occurs because of three effects:

- distributed pressure losses along the pipe's inner surface
- local pressure losses at distinct items,
- hydrostatic pressure differences because of a difference in height.

All three effects can be captured in this formula:

$$\Delta p = \Delta p_{loc} + \Delta p_{dis} + \Delta p_{hydrostatic}$$

Distributed pressure losses

The Darcy-Weissbach-equation describes distributed pressure losses Δp_{dis} inside the pipe:

$$\Delta p_{dis} = \lambda \rho \frac{L}{2D} v^2.$$

Together with the flow velocity

$$v = \frac{4\dot{m}}{\rho \pi D^2}$$

it can be written to:

$$\Delta p_{dis} = \lambda \frac{8L}{\rho \pi^2 D^5} \dot{m}^2,$$

where the darcy friction factor $\lambda = \lambda(Re, \epsilon, D)$ depends on the Reynolds number Re :, the pipe's surface roughness ϵ and the pipe's inner diameter D. The Reynolds number is a dimensionless quantity characterizing fluid flows and is defined as follows:

$$Re = \frac{Dv\rho}{\mu}.$$

μ is the dynamic viscosity of water.

In a pipe, flow is laminar if Re < 2300 and turbulent if Re > 4000. In district heating pipes, flow is usually turbulent. The turbulent flow regime can be further distinguished into smooth, intermediate and rough regime depending on the pipe's surface roughness.

[1] provides the following approximation formula for λ:

$$\lambda = 0.07 \cdot Re^{-0.13} \cdot D^{-0.14}.$$

A more accurate approximation of the Colebrook-White-equation for flow in pipes is given by this formula:

$$\lambda = \frac{1.325}{(ln(\frac{\epsilon}{3.7D} + \frac{5.74}{Re^{0.9}}))^2}.$$

Local pressure losses

Local pressure losses are losses at junction elements, angles, valves etc. They are described by the localized pressure loss coefficient ζ:

$$\Delta p_{loc} = \zeta \frac{v^2}{2} \rho$$

It is assumed that each fork has a tee installed. According to [2], localized pressure losses occur downstream of the element that causes these losses. The values of the localized pressure loss coefficient ζ were taken from [3]. In case of a tee which splits the stream, ζ is 2. In case the streams join, ζ is 0.75.

It is also assumed that each consumer has a valve installed. Due to the complexity of determining the localized pressure loss coefficients, these losses have not been considered so far.

Hydrostatic pressure difference

The hydrostatic pressure difference is calculated as follows:

$$\Delta p_{hydrostatic} = -\rho g \Delta h$$

Pump power

The mass flow in the pipes is driven by the pressure difference that is generated by pumps. The pumps have to balance the pressure losses inside the pipes. The pump power thus depends on the pressure difference along the inlet and return along one strand of the network, Δp, the mass flow $\dot{m}$ and the pump's efficiency $\eta_{pump} = \eta_{el} \cdot \eta_{hyd}$.

$$P_{el.pump} = \frac{1}{\eta_{el}\eta_{hyd}} \frac{\Delta p}{\rho} \dot{m}$$

In a network consisting of several strands, the strand with the largest pressure losses in inlet and return defines the pressure difference that the pumps have to generate. The underlying assumption is that the consumers at the end of all other strands adjust their valve to generate the same pressure losses such that the mass flows that are assumed are met.

5.3.2 Thermal equations

The temperature spread between inlet and return flow defines the amount of heat that is transported with a given mass flow:

$$\dot{Q} = \dot{m} \cdot c \cdot \Delta T.$$

A larger temperature spread allows smaller pipe's diameters, which reduces the investment cost of new pipes or increases the thermal power of existing pipes.

Heat losses

Heat losses depend on temperature level, mass flow and pipe insulation. Especially the representation of the heat losses depends a lot on the level of detail of a model. As mentioned above, in the current implementation, the thermal state of the network is assumed to be in steady state conditions. The temperature at the outlet is calculated as follows:

$$T_{out} = T_{env} + (T_{in} - T_{env}) \cdot exp\{-\frac{U\pi DL}{c \cdot \dot{m}}\}.$$

Where T_{in} and T_{out} are the temperatures at the start and end of the pipe, T_{env} the environmental temperature and U the thermal transmittance.

In data documentation of pipes in a district heating, you often find the value of the specific heat loss per meter $U_{spec}[W/(Km)]$.

$$\begin{aligned} U_{spec} &= U \cdot \pi D && \text{for single pipes} \\ U_{spec} &= U \cdot 2\pi D && \text{for double pipes} \end{aligned}$$

The temperature of the return flow at the fork is calculated assuming ideally mixed flows, where no heat losses occur and the heat capacity is constant. The temperature of the mixed flow T_{mix} is calculated for a number n of inlet flows, that are ideally mixed, using the following equation:

$$T_{mix} = \frac{\sum_{j=1}^{n} (\dot{m}_n \cdot T_n)}{\dot{m}_{mix}}$$

5.4 References

CHAPTER 6

Model coupling

CHAPTER 7

Analyzing and plotting results

CHAPTER 8

What's New

These are new features and improvements of note in each release

Releases

8.1 v0.0.1 (28.10.2020)

First release by the oemof developing group.

8.1.1 Contributors

- Jann Launer
- Johannes Röder
- Joris Zimmermann
- Marie-Claire Gering
- oakca

CHAPTER 9

API Documentation

9.1 network

This module is designed to hold the definition of the central ThermalNetwork object and its components.

This file is part of project dhnx (). It's copyrighted by the contributors recorded in the version control history of the file, available from its original location:

SPDX-License-Identifier: MIT

class dhnx.network.**ThermalNetwork**(*dirname=None*)

Bases: object

Class representing thermal (heating/cooling) networks.

Parameters

- **availalable_components** –
- **component_attrs** –
- **components** –
- **sequences** –
- **results** –
- **graph** –

Examples

```
>>> from dhnx.network import ThermalNetwork
>>> tnw = ThermalNetwork('csv_folder')
>>> tnw.is_consistent()
True
```

add(*class_name*, *id*, ***kwargs*)
Adds a row with id to the component DataFrame specified by class_name.

Parameters

- **class_name** –
- **id** –
- **kwargs** –

from_csv_folder(*dirname*)

is_consistent()

Checks that

- pipes connect to existing nodes,
- pipes do not connect a node with itself,
- there are no duplicate pipes between two nodes.

optimize_investment(*invest_options*, ***kwargs*)

optimize_operation()

remove(*class_name*, *id*)
Removes the row with id from the component DataFrame specified by class_name.

Parameters

- **class_name** (*str*) – Name of the component class
- **id** (*int*) – id of the component to remove

reproject(*crs*)

set_defaults()
Sets default values on component DataFrames.

Returns

Return type None

set_timeindex()
Takes all sequences and checks if their timeindex is identical. If that is the case, it sets the timeindex attribute of the class. If there are no sequences given, the timeindex will keep the default value.

simulate(**args*, ***kwargs*)

to_csv_folder(*dirname*)

to_nx_graph()

9.2 model

This module is designed to base classes for optimization and simulation models.

This file is part of project dhnx (). It's copyrighted by the contributors recorded in the version control history of the file, available from its original location:

SPDX-License-Identifier: MIT

class dhnx.model.**InvestOptimizationModel**(*thermal_network*)
Bases: *dhnx.model.Model*

Abstract base class for investment optimization models.

is_consistent()

class dhnx.model.**Model**(*thermal_network*)
Bases: object

Abstract base class for different kind of models.

get_results()

is_consistent()

setup()

solve()

class dhnx.model.**OperationOptimizationModel**(*thermal_network*)
Bases: *dhnx.model.Model*

Abstract base class for operational optimization models.

is_consistent()

class dhnx.model.**SimulationModel**(*thermal_network*)
Bases: *dhnx.model.Model*

Abstract base class for simulation models.

is_consistent()

9.3 optimization

This module is designed to hold optimization model implementations. The implementation makes use of oemof-solph.

This file is part of project dhnx (). It's copyrighted by the contributors recorded in the version control history of the file, available from its original location:

SPDX-License-Identifier: MIT

class dhnx.optimization.**OemofInvestOptimizationModel**(*thermal_network*, *settings*, *investment_options*)
Bases: *dhnx.model.InvestOptimizationModel*

Implementation of an invest optimization model using oemof-solph.

...

settings
Dictionary holding the optimisation settings. See .

Type dict

invest_options
Dictionary holding the investment options for the district heating system.

Type dict

nodes
Empty list for collecting all oemof.solph nodes.

Type list

buses
Empty dictionary for collecting all oemof.solph.Buses of the energy system.

Type dict

es
Empty oemof.solph.EnergySystem.

Type oemof.solph.EnergySystem

om
Attribute, which will be the oemof.solph.Model for optimisation.

Type oemof.solph.Model

oemof_flow_attr
Possible flow attributes, which can be used additionally: {'nominal_value', 'min', 'max', 'variable_costs', 'fix'}

Type set

results
Empty dictionary for the results.

Type dict

check_input():
Performs checks on the input data.

complete_exist_data():
Sets the investment status for the results dataframe of the pipes.

get_pipe_data():
Adds heat loss and investment costs to pipes dataframe.

setup_oemof_es():
The energy system *es* is build.

setup():
Calls *check_input()*, *complete_exist_data()*, *get_pipe_data()*, and *setup_oemof_es()*.

check_existing()
Checks if the attributes *existing* and *hp_type* are given in the *pipes* table. If not, the attribute is added, and set to *None* / 0.

Checks for all existing pipes, if the heatpipe type is given in the pipe type table *.invest_options['network']['pipes']*, and if the capacity is greater than zero.

check_input()
Check 1:

Check and make sure, that the dtypes of the columns of the sequences and the indices (=ids) of the forks, pipes, producers and consumers are of type 'str'. (They need to be the same dtye.)

Check 2:

Firstly, it is checked, if there are any not-allowed connection in the *pipe* data. The following connections are not allowed:

- consumer -> consumer
- producer -> producer
- producer -> consumer
- consumer -> fork

Secondly, it is checked, if a pipes goes to a consumer, which does not exist.

Check 3

Checks if graph of network is connected.

An error is raised if one of these connection occurs.

get_results_edges()
Postprocessing of the investment results of the pipes.

prepare_heat_demand()
This method performs the pre-processing of the heat demand data, depending on the given optimisation settings.

- If attribute 'P_heat_max' not given at the consumers, the maximum heat demand is calculated from the timeseries and added the consumers table.
- If the optimisation setting 'heat_demand' == scalar, the number of time steps of the optimisation is set to 1, and the 'P_heat_max' values are copied to the consumers heat flow sequences (which is always the input for the optimisation model).
- The consumers heat flow sequences are multiplied by the simultaneity factor.
- Finally, a sufficient length of the heat demand timeseries is checked.

Returns

- Updated *.network.components['consumers']* and
- *.network.sequences['consumers']['heat_flow']*

remove_inactive()
If the attribute active is present in any of the components columns, or in any the investment options tables, all rows with active == 0 are deleted, and the column active is deleted.

setup()
Calls *remove_inactive()* *check_input()*, *prepare_heat_demand()*, *complete_exist_data()*, and *setup_oemof_es()*.

setup_oemof_es()
The oemof solph energy system is initialised based on the settings, and filled with oemof-solph object:

The oemof-solph objects of the *consumers* and *producers* are defined at the consumers and producers investment options.

For the heating infrastructure, there is a *oemof.solph.Bus* added for every fork, and a pipe component for every pipe as defined in */network/pipes.csv*.

solve()
Builds the oemof.solph.Model of the energysystem *es*.

class dhnx.optimization.**OemofOperationOptimizationModel**(*thermal_network*)
Bases: *dhnx.model.OperationOptimizationModel*

Implementation of an operation optimization model using oemof-solph.

get_results()

setup()

solve()

dhnx.optimization.**optimize_operation**(*thermal_network*)
Takes a thermal network and returns the result of the operational optimization.

dhnx.optimization.**setup_optimise_investment**(*thermal_network*, *invest_options*, *heat_demand='scalar'*, *num_ts=1*, *time_res=1*, *start_date='1/1/2018'*, *frequence='H'*, *solver='cbc'*, *solve_kw=None*, *solver_cmdline_options=None*, *simultaneity=1*, *bidirectional_pipes=False*, *dump_path=None*, *dump_name='dump.oemof'*, *print_logging_info=False*, *write_lp_file=False*)

Function for setting up the oemof solph operational Model.

Parameters

- **thermal_network** (ThermalNetwork) – See the ThermalNetwork class.
- **invest_options** (*dict*) – Dictionary holding the investment options for the district heating system.
- **heat_demand** (*str*) – 'scalar': Peak heat load is used as heat consumers' heat demand. 'series': Heat load time-series is used.
- **num_ts** (*int*) – Number of time steps of optimisation.
- **time_res** (*float*) – Time resolution.
- **start_date** (*str or datetime-like*) – Startdate for oemof optimisation.
- **frequence** (*str or DateOffset*) – Lenght of period.
- **solver** (*str*) – Name of solver.
- **solve_kw** (*dict*) – Solver kwargs.
- **solver_cmdline_options** (*dict*) – Dictionary with command line options for solver.
- **simultaneity** (*float*) – Simultaneity factor.
- **bidirectional_pipes** (*bool*) – Bidirectional pipes leads to bi-directional flow attributes at the heatpipeline components {'min': -1, bidirectional: True}.
- **dump_path** (*str*) – If a dump path is provided, the oemof dump file is stored.
- **dump_name** (*str*) – Name of dump file.
- **print_logging_info** (*bool*) – Additional logging info is printed.
- **write_lp_file** (*bool*) – Linear program file is stored ('User/.oemof/lp_files/DHNx.lp').

Returns **oemof.solph.Model**

Return type The oemof.solph.Model is build.

dhnx.optimization.**solve_optimisation_investment**(*model*)

Parameters **model** (*oemof.solph.Model*) – The oemof model, which is optimized.

Returns

dict –

- 'oemof' : Complete "oemof" results of the energy system optimisation (.results['main']).
- 'oemof_meta' : Meta results of oemof solph optimisation.

- 'components' : 'pipes' : Investment results of pipes.

Return type Results of optimisation. Contains:

9.4 simulation

This module is designed to hold implementations of simulation models. The implementation uses oemof/tespy.

This file is part of project dhnx (). It's copyrighted by the contributors recorded in the version control history of the file, available from its original location:

SPDX-License-Identifier: MIT

class dhnx.simulation.**SimulationModelNumpy**(*thermal_network*, *rho=971.78*, *c=4190*, *mu=0.00035*, *eta_pump=1*, *tolerance=1e-10*)

Bases: *dhnx.model.SimulationModel*

Implementation of a simulation model using numpy.

prepare()

solve()

get_results()

prepare_hydraulic_eqn()
Prepares the input data for the hydraulic problem.

prepare_thermal_eqn()
Prepares the input data for the thermal problem.

solve_hydraulic_eqn()
Solves the hydraulic problem.

solve_thermal_eqn()
Solves the thermal problem.

_concat_scalars(*name*)
Concatenates scalars of all components with a given variable name

Parameters **name** (*str*) – Name of the variable

Returns **concat_sequences** – DataFrame containing the sequences

Return type pd.DataFrame

_concat_sequences(*name*)
Concatenates sequences of all components with a given variable name

Parameters **name** (*str*) – Name of the variable

Returns **concat_sequences** – DataFrame containing the sequences

Return type pd.DataFrame

static _set_producers_mass_flow(*m*)
Sets the mass flow of the producer.

Parameters **m** (*pd.DataFrame*) – DataFrame with all know consumer mass flows.

Returns **m** – DataFrame with all know mass flow of consumers and producer.

Return type pd.DataFrame

_calculate_pipes_mass_flow()
Determines the mass flow in all pipes using numpy's least squares function.

Returns **pipes_mass_flow** – Mass flow in the pipes [kg/s]

Return type pd.DataFrame

_calculate_reynolds()
Calculates the Reynolds number.

$$Re = \frac{4\dot{m}}{\pi \mu D}$$

Returns **re** – Reynolds number for every time step and pipe [-]

Return type pd.DataFrame

_calculate_lambda(*reynolds*)
Calculates the darcy friction factor.

$$\lambda = 0.07 \cdot Re^{-0.13} \cdot D^{-0.14}$$

Parameters **re** (*pd.DataFrame*) – Reynolds number for every time step and pipe [-]

Returns **lamb** – Darcy friction factor for every time step and pipe [-]

Return type pd.DataFrame

_calculate_pipes_distributed_pressure_losses(*lamb*)
Calculates the pressure losses in the pipes.

Equal-sized inlet and return pipes are assumed which leads to equal mass flows and pressure losses for both. This introduces the initial factor of 2 in the equation.

$$\delta p = 2 \cdot \lambda \frac{8L}{\rho \pi^2 D^5} \dot{m}^2.$$

Parameters **lamb** (*pd.DataFrame*) – Darcy friction factor for every time step and pipe [-]

Returns **pipes_pressure_losses** – DataFrame with distributed pressure losses for inlet and return for every time step and pipe [Pa]

Return type pd.DataFrame

_calculate_pipes_localized_pressure_losses()
Calculates localized pressure losses at the nodes.

$$\Delta p_{loc} = \frac{8\zeta \dot{m}^2}{\rho \pi^2 D^4}$$

Returns **nodes_pressure_losses** – Localized pressure losses at the nodes [Pa]

Return type pd.DataFrame

_calculate_global_pressure_losses(*pipes_pressure_losses*)
Calculates global pressure losses.

Finds the path with the maximal pressure loss among from the set of paths from the producer to all consumers.

Parameters **pipes_pressure_losses** (*pd.DataFrame*) – Total pressure losses for every time step and pipe [Pa]

Returns **global_pressure_losses** – Global pressure losses [Pa]

Return type pd.DataFrame

_calculate_pump_power(*global_pressure_losses*)
Calculates the pump power.

$$P_{el.pump} = \frac{1}{\eta_{el}\eta_{hyd}} \frac{\Delta p}{\rho} \dot{m}$$

Parameters **global_pressure_losses** (*pd.DataFrame*) – Global pressure losses [Pa]

Returns **pump_power** – Pump power [W]

Return type pd.Series

_calculate_exponent_constant()
Calculates the constant part of the exponent that determines the cooling of the medium in the pipes.

$$exp_{const} = -\frac{U\pi DL}{c}$$

Returns **exponent_constant** – Constant part of the exponent [kg/s]

Return type np.matrix

_calc_temps(*exponent_constant*, *known_temp*, *direction*)
Calculate temperatures

$$T_{out} = T_{env} + (T_{in} - T_{env}) \cdot exp\{exp_{const} \cdot exp_{var}\} = T_{out} = T_{env} + (T_{in} - T_{env}) \cdot exp\{-\frac{U\pi DL}{c \cdot \dot{m}}\}$$

Parameters

- **exponent_constant** (*np.array*) – Constant part of the exponent [kg/s]
- **known_temp** (*pd.DataFrame*) – Known temperatures at producers or consumers [°C]
- **direction** (*+1 or -1*) – For inlet and return flow [-]

Returns **temp_df** – DataFrame containing temperatures for all nodes [°C]

Return type pd.DataFrame

_set_temp_return_input(*temp_inlet*)
Sets the temperature of the return pipes at the consumers.

$$T_{cons,r} = T_{cons,i} - T_{cons,drop}$$

Parameters **temp_inlet** (*pd.DataFrame*) – Known inlet temperature [°C]

Returns **temp_return** – Return temperature with the consumers values set [°C]

Return type pd.DataFrame

_calculate_pipes_heat_losses(*temp_node*)
Calculates the pipes' heat losses given the temperatures.

$$\dot{Q}_{losses} = c \cdot \dot{m} \cdot \Delta T$$

Parameters **temp_node** (*pd.DataFrame*) – Temperatures at the nodes [°C]

Returns **pipes_heat_losses** – Heat losses in the pipes [W]

Return type pd.DataFrame

dhnx.simulation.**simulate**(*thermal_network*, *results_dir=None*)
Takes a thermal network and returns the result of the simulation.

Parameters **thermal_network** –

Returns **results**

Return type dict

CHAPTER 10

Authors

–**alphabetic order**–

- Jann Launer
- Johannes Röder
- Joris Zimmermann
- Marie-Claire Gering
- oakca

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

T