DHNx

Release 0.0.3.dev0

Dec 14, 2022

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>
```

Note: DHNx uses geopandas and osmnx as extra requirements for some functions related to the processing of spatial data. On Windows machines, you might encounter troubles installing geopandas via `pip install geopandas`. Try to install geopandas in an EMTPY environment with `conda install geopandas`, first. And second, install osmnx with `pip install osmnx` (tested with Python 3.8). Also check geopandas.org.

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.

For testing, *tox* is used. Either run simply *tox* for all checks, or specify your test run depending on your local python environment, e.g. by:

```
tox -e "clean, check, docs, py38"
```

As DHNx is part of the oemof developer group we use the same developer rules. You will find more information in the oemof meta documentation.

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='producers-0', to_node='consumers-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

Geometry preparation

For setting up a *ThermalNetwork*, you need to prepare your input data somehow. Therefore, the dhnx package provides some helpful modules and function with the geometry processing modules, see *connect_points*, especially the *dhnx.gistools.connect_points.process_geometry()*.

The example folder included in this repository also contains the *import_osm_invest* example, that provides an illustrative introduction on how to use and prepare your geometry based on open street maps data (See `examples/investment_optimisation/import_osm_invest`).

CHAPTER 5

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.

5.1 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.

5.2 Usage

Links to the subsections:

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

5.2.1 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.

5.2.2 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. With *nonconvex* set to 1, fix losses and fix costs independent of the dimension of the pipelines capacity can be considered. It is recommended to set *nonconvex* to 1, as the construction of DHS pipelines is usually characterized by a high share of fixed costs. If *nonconvex* is 0, the costs-curve is a line through origin.

- **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. Recommended to use with *nonconvex* is *True*.
- **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. [€]) This attribute requires *nonconvex* is set to *True*.

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

The module *dhnx.optimization.precalc_hydraulic* provides helpful functions for the calculation of the pipeline parameters (see *dhnx.optimization.precalc_hydraulic*). See also the precalulation example in the example folder of the optimisation examples.

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'*.

5.2.3 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.

5.2.4 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*.

5.3 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 6

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.

6.1 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.

6.2 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
```

6.3 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.

6.3.1 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.

6.3.2 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:

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

6.4 References

CHAPTER 7

Model coupling

CHAPTER 8

Analyzing and plotting results

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()
>>> 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 *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 gistools

9.2.1 dhnx.gistools.geometry_operations

This modules holds functions for geometry operations, that are needed for the geometry processing module *connect_points.py*.

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: https://github.com/oemof/DHNx

This module is not fully tested yet, so use it with care.

SPDX-License-Identifier: MIT

dhnx.gistools.geometry_operations.**any_check**(*geom_test*, *gdf*, *how*)

Improve speed for an 'any()' test on a list comprehension.

Replace a statement like...

```
if any([geom_test.touches(g) for g in gdf.geometry]):
```

... with the following:

```
if any_check(geom_test, gdf, how='touches'):
```

Instead of iterating through all of 'g in gdf.geometry', return 'True' after the first match.

Parameters

- **geom_test** (*Shapely object*) – Object which's function 'how' is called.
- **gdf** (*GeoDataFrame*) – All geometries in gdf are passed to 'how'.
- **how** (*str*) – Shapely object function like equals, almost_equals, contains, crosses, disjoint, intersects, touches, within.

Returns *bool* – True if any call of function 'how' is True.

dhnx.gistools.geometry_operations.**check_crs**(*gdf*, *crs=4647*)

Convert CRS to EPSG:4647 - ETRS89 / UTM zone 32N (zE-N).

This is the (only?) Coordinate Reference System that gives the correct results for distance calculations.

dhnx.gistools.geometry_operations.**check_double_points**(*gdf*, *radius=0.001*, *id_column=None*)

Check for points, which are close to each other.

In case, two points are close, the index of the points are printed.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame with Points as geometry.
- **radius** (*float*) – Maximum distance.
- **id_column** (*str or None*) – Column name which should be printed in case of near points. If None, the index is printed.

Returns **list** (*Indices of "near" points.*)

dhnx.gistools.geometry_operations.**create_forks**(*lines*)

Creates a forks(nodes) GeoDataFrame from a "line"-GeoDataFrame based on the end-points of each LineString.

Also, an index for every fork is given, and the columns 'full-id' (="forks-" + index"), 'lat' and 'lon', which results from the geometry, are added to the GeoDataFrame.

Parameters **lines** (*geopandas.GeoDataFrame*)

Returns **geopandas.GeoDataFrame** (*GeoDataFrame with Points as geometry.*)

dhnx.gistools.geometry_operations.**gdf_to_df**(*gdf*)

Converts a GeoDataFrame to a pandas.DataFrame by deleting the geometry column.

dhnx.gistools.geometry_operations.**insert_node_ids**(*lines*, *nodes*)

Creates the columns *from_node*, *to_node* and inserts the node ids (eg. forks-3, consumers-5). The updated "line"-GeoDataFrame is returned.

Parameters

- **lines** (*geopandas.GeoDataFrame*)
- **nodes** (*geopandas.GeoDataFrame*)

Returns *geopandas.GeoDataFrame*

dhnx.gistools.geometry_operations.**split_multilinestr_to_linestr**(*gdf_input*)
Simplifies GeoDataFrames with LineStrings as geometry.

The LineStrings (whether LineStrings, or MulitLineStings) are split into LineStrings with only two coordinates, one starting and one ending point.

The other values of the GeoDataFrame are copied to the new rows for each row, who's geometry is split.

Parameters **gdf_lines** (*geopandas.GeoDataFrame*)

Returns *geopandas.GeoDataFrame*

dhnx.gistools.geometry_operations.**weld_segments**(*gdf_line_net*, *gdf_line_gen*, *gdf_line_houses*, *debug_plotting=False*)
Weld continuous line segments together and cut loose ends.

This is a public function that recursively calls the internal function weld_line_segments_(), until the problem cannot be simplified further.

Find all lines that only connect to one other line and connect those to a single MultiLine object. Points that connect to Generators and Houses are not simplified. Loose ends are shortened where possible.

Parameters

- **gdf_line_net** (*GeoDataFrame*) – Potential pipe network.
- **gdf_line_gen** (*GeoDataFrame*) – Generators that need to be connected.
- **gdf_line_houses** (*GeoDataFrame*) – Houses that need to be connected.
- **debug_plotting** (*bool, optional*) – Plot the selection process.

Returns **gdf_line_net_new** (*GeoDataFrame*) – Simplified potential pipe network.

9.2.2 dhnx.gistools.connect_points

This module holds functions for processing the geometry for setting up the geometry of a ThermalNetwork based on a street geometry and a table of buildings.

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: https://github.com/oemof/DHNx

This module is not fully tested yet, so use it with care.

SPDX-License-Identifier: MIT

dhnx.gistools.connect_points.**calc_lot_foot**(*line*, *point*)
Calculates the lot foot point.

Parameters

- **line** (*shapely.geometry.LineString*)
- **point** (*shapely.geometry.Point*)

Returns *shapely.geometry.Point*

dhnx.gistools.connect_points.**check_duplicate_geometries**(*gdf*)
Test the input GeoDataFrame for duplicate geometries and plot them.

dhnx.gistools.connect_points.**check_geometry_type**(*gdf*, *types*)
Checks, if a geodataframe has only the given geometry types in its GeoSeries.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – DataFrame to be checked.
- **types** (*list*) – List of types allowed for GeoDataFrame.

dhnx.gistools.connect_points.**create_object_connections**(*points*, *lines*, *tol_distance=1*)
Connects points to a line network.

Generally, the nearest point of the next line is used as connection the point. Depending on the geometry, there are 3 options, the connection is created:

- nearest point is line ending => the connection line starts from this line ending
- nearest point is on the next line:

  a) line endings are outside the tolerance => line is split and the nearest point is used as connection point

  b) line endings are within the tolerance distance => the next line ending is used as connection point

The tolerance distance avoids the generation of short line elements. This is for example the case if two buildings are directly opposite of the street. Using simply the nearest point method could result in very short lines.

Parameters

- **points** (*geopandas.GeoDataFrame*) – Points which should be connected to the line. GeoDataFrame with Points as geometry.
- **lines** (*geopandas.GeoDataFrame*) – The line-network to which the Points should be connected. The line geometry needs to consists of simple lines based on one starting and one ending point. LineStrings which contain more than 2 points are not allowed.
- **tol_distance** (*float*) – Tolerance distance for choosing the end of the line instead of the nearest point.

Returns

- **geopandas.GeoDataFrame** (*The newly created connection lines*)
- **geopandas.GeoDataFrame** (*The updated lines (some lines are split.)* – All lines should only touch at the line endings.

dhnx.gistools.connect_points.**create_points_from_polygons**(*gdf*, *method='midpoint'*)
Converts the geometry of a polygon layer to a point layer.

Parameters

- **gdf** (*geopandas.GeoDataFrame*)
- **method** (*str*) – Method to create a point from a polygon.

Returns **geopandas.GeoDataFrame** (*GeoDataFrame with a point geometry.*)

dhnx.gistools.connect_points.**line_of_point**(*point*, *gdf_lines*)

Gets index of geometry of a GeoDataFrame, a point is located next to, with a distance lower than 1e-8.

Parameters

- **point** (*shapely.geometry.Point*)
- **gdf_lines** (*geopandas.GeoDataFrame*)

Returns **int, float or str** (*Index of GeoDataFrame or Warning, if no geometry found.*)

dhnx.gistools.connect_points.**point_to_array**(*point*)

Returns the coordinates of a point as numpy.array

Parameters **point** (*shapely.geometry.Point*)

Returns *numpy.array()*

dhnx.gistools.connect_points.**process_geometry**(*lines*, *consumers*, *producers*, *method='midpoint'*, *projected_crs=4647*, *tol_distance=2*, *reset_index=True*)

This function connects the consumers and producers to the line network, and prepares the attributes of the geopandas.GeoDataFrames for importing as dhnx.ThermalNetwork.

The ids of the lines are overwritten.

Parameters

- **lines** (*geopandas.GeoDataFrame*) – Potential routes for the DHS. Expected geometry Linestrings or MultilineStrings. The graph of this line network should be connected.
- **consumers** (*geopandas.GeoDataFrame*) – Location of demand/consumers. Expected geometry: Polygons or Points.
- **producers** (*geopandas.GeoDataFrame*) – Location of supply sites. Expected geometry: Polygons or Points.
- **method** (*str*) – Method for creating the point if polygons are given for the consumers and producers. Method 'midpoint' uses the centroid of each building polygon. Method 'boundary' moves the point to the boundary (wall) of the building, along the line constructed from centroid to the street.
- **multi_connections** (*bool*) – Setting if a building should be connected to multiple streets.
- **projected_crs** (*EPSG integer code*) – EPSG Coordinate reference system number (eg 4647), which is used for the geometry operations. A projected crs must be used!
- **tol_distance** (*float*) – Tolerance distance at connection the points to the line network for choosing the end of the line instead of the lot.
- **reset_index** (*bool*) – If True, reset the index and ignore the existing index. If False, use the existing index for consumer and producer identificators. Default: True

Returns **dict** (*Dictionary with 4 geopandas.GeoDataFrames: The keys of the Dict are*) – equal to the components of the dhnx.ThermalNetwork: 'forks', 'consumers', 'producers', 'pipes'.

dhnx.gistools.connect_points.**run_point_method_boundary**(*consumers_poly*, *consumers*, *producers_poly*, *producers*, *lines_consumers*, *lines_producers*)

Run 'boundary' method for finding the building connection point.

The 'midpoint' method (using the centroid) must already have been run, generating the default connection lines from street to centroid.

If there is only one intersection between that line and the boundary of the building, this intersection point is used as the connection point instead (and the connection line is shortened accordingly).

However, complex building shapes can produce multiple intersections. In this case, the intersection with the 'convex hull' of the building is used instead. This may result in connection points that do not touch an actual building wall, but it should still be an improvement compared to the 'midpoint' method.

In case of no intersections with the building boundary (possible for e.g. U-shaped buildings), the original centroid is used.

Parameters

- **consumers_poly** (*geopandas.GeoDataFrame*) – Polygons of the consumer buildings. Point geometries are also allowed, but they are not changed.
- **consumers** (*geopandas.GeoDataFrame*) – Points of the consumer buildings (as returned by 'midpoint' method).
- **producers_poly** (*geopandas.GeoDataFrame*) – Polygons of the producer buildings. Point geometries are also allowed, but they are not changed.
- **producers** (*geopandas.GeoDataFrame*) – Points of the producer buildings (as returned by 'midpoint' method).
- **lines_consumers** (*geopandas.GeoDataFrame*) – Connection lines from street to each consumer point.
- **lines_producers** (*geopandas.GeoDataFrame*) – Connection lines from street to each producer point.

Returns

- **consumers** (*geopandas.GeoDataFrame*) – Updated points of the consumer buildings.
- **producers** (*geopandas.GeoDataFrame*) – Updated points of the producer buildings.
- **lines_consumers** (*geopandas.GeoDataFrame*) – Updated connection lines from street to each consumer point.
- **lines_producers** (*geopandas.GeoDataFrame*) – Updated connection lines from street to each producer point.

9.3 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()

9.4 optimization

9.4.1 dhnx.optimization.optimization_models

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

Implementation of an invest optimization model using oemof-solph.

...

Variables

- **settings** (*dict*) – Dictionary holding the optimisation settings. See .
- **invest_options** (*dict*) – Dictionary holding the investment options for the district heating system.
- **nodes** (*list*) – Empty list for collecting all oemof.solph nodes.
- **buses** (*dict*) – Empty dictionary for collecting all oemof.solph.Buses of the energy system.
- **es** (*oemof.solph.EnergySystem*) – Empty oemof.solph.EnergySystem.
- **om** (*oemof.solph.Model*) – Attribute, which will be the oemof.solph.Model for optimisation.
- **oemof_flow_attr** (*set*) – Possible flow attributes, which can be used additionally: {'nominal_value', 'min', 'max', 'variable_costs', 'fix'}
- **results** (*dict*) – Empty dictionary for the results.

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.optimization_models.**OemofOperationOptimizationModel**(*thermal_network*)
Bases: *dhnx.model.OperationOptimizationModel*

Implementation of an operation optimization model using oemof-solph.

get_results()

setup()

solve()

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

dhnx.optimization.optimization_models.**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** (*The oemof.solph.Model is build.*)

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

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

Returns

dict (*Results of optimisation. Contains:*) –

- '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.

9.4.2 dhnx.optimization.dhs_nodes

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

dhnx.optimization.dhs_nodes.**add_nodes_dhs**(*opti_network*, *gd*, *nodes*, *busd*)

Based on the *forks* and *pipes* of the *ThermalNetwork* of the *OemofInvestOptimisationModel*, the oemof-solph components for the district heating network optimisation are initialised.

For all *forks*, a *solph.Bus* with the label *infrastructure_heat_bus_forks-<id>* (string representation) is generated.

For all *pipes*, a 'HeatPipeline' with the label *infrastructure_heat_<type>_<from_node>-<to_node>* (string representation) is generated. Depending on the attributes in *pipes*, an investment pipe, or an existing pipe is built.

Parameters

- **opti_network** (*OemofInvestOptimisationModel*)
- **gd** (*dict*) – General optimisation settings.
- **nodes** (*list*) – List for collecting all oemof-solph objects.
- **busd** (*dict*) – Dictionary collecting all oemof-solph Buses. Keys: labels of the oemof-solph Buses; Values: oemof-solph Buses.

Returns **list, dict** (*List of all oemof-solph objects and dictionary of oemof-solph Buses.*)

dhnx.optimization.dhs_nodes.**add_nodes_houses**(*opti_network*, *nodes*, *busd*, *label_1*)

For each *consumers/producers* of the *ThermalNetwork* of the *OemofInvestOptimisationModel*, the oemof-solph components for the *consumers/producers* are initialised depending on the given tables of the `invest_options` of the *OemofInvestOptimisationModel*.

The tables of *invest_options* provide the information and attributes for the oemof-solph components, which should be build at every *consumer/producer*.

The minimum requirement is to provide table with at least one *heat Bus* for all consumers and all producers, a table with a *heat sink* as demand for all consumers, and a table with at least one *heat source* for all producers.

Additionally, further tables with *Transformer*, *Storages*, and further *Sources* and *Sinks* can be added.

For the attributes for each table, you need to provide, please see:

Parameters

- **opti_network** (*OemofInvestOptimisationModel*)
- **gd** (*dict*) – General optimisation settings.
- **nodes** (*list*) – List for collecting all oemof-solph objects.
- **busd** (*dict*) – Dictionary collecting all oemof-solph Buses. Keys: labels of the oemof-solph Buses; Values: oemof-solph Buses.
- **label_1** (*str*) – First tag of the label, which is either *producers* or *consumers*.

Returns **list, dict** (*List of all oemof-solph objects and dictionary of oemof-solph Buses.*)

9.4.3 dhnx.optimization.add_components

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

dhnx.optimization.add_components.**add_buses**(*it*, *labels*, *nodes*, *busd*)

This function initialises the oemof.solph.Bus classes of the energysystem based on the given tabular information. Additionally, a sink or a source can be added to every bus and costs for this source (shortage) or sink (excess) can be defined.

The table must be given as follows:

Table 1: Example for table of *Buses*

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

Parameters

- **it** (*pd.DateFrame*) – Table of attributes for Buses for the producers and consumers.
- **labels** (*dict*) – Dictonary containing tag1 and tag4 of label-tuple.
- **nodes** (*list*) – All oemof.solph components are added to the list.
- **busd** (*dict*) – All buses are added to this dictionary.

Returns **list, dict** (*Updated list of nodes and dict of Buses.*)

dhnx.optimization.add_components.**add_demand**(*it*, *labels*, *series*, *nodes*, *busd*)
Initialisation of oemof.solph.Sinks which represent demands.

Parameters

- **it** (*pd.DataFrame*) – Table of attributes for Sources for the producers and consumers.
- **labels** (*dict*) – Dictonary containing specifications for label-tuple.
- **series** (*dict*) – Contain the heat demand time-series of all consumers.
- **nodes** (*list*) – All oemof.solph components are added to the list.
- **busd** (*dict*) – All oemof.solph.Bus objects are given by this dictionary.

Returns **list** (*Updated list of nodes.*)

dhnx.optimization.add_components.**add_heatpipes**(*it*, *labels*, *bidirectional*, *length*, *b_in*, *b_out*, *nodes*)
Adds *HeatPipeline* objects with *Investment* attribute to the list of oemof.solph components.

Parameters

- **it** (*pd.DataFrame*) – Table of *Heatpipeline* attributes of the district heating grid
- **labels** (*dict*) – Dictonary containing specifications for label-tuple
- **bidirectional** (*bool*) – Settings for creating bidirectional heatpipelines
- **length** (*float*) – Length of pipeline
- **b_in** (*oemof.solph.Bus*) – Bus of Inflow
- **b_out** (*oemof.solph.Bus*) – Bus of Outflow
- **nodes** (*list*) – All oemof.solph components are added to the list

Returns **list** (*Updated list of nodes.*)

dhnx.optimization.add_components.**add_heatpipes_exist**(*pipes*, *labels*, *gd*, *q*, *b_in*, *b_out*, *nodes*)
Adds *HeatPipeline* objects with fix capacity for existing pipes to the list of oemof.solph components.

Parameters

- **pipes**
- **labels** (*dict*) – Dictonary containing specifications for label-tuple.
- **gd** (*dict*) – Settings of the investment optimisation of the ThermalNetwork
- **q** (*pd.Series*) – Specific *Pipe* of ThermalNetwork
- **b_in** (*oemof.solph.Bus*) – Bus of Inflow
- **b_out** (*oemof.solph.Bus*) – Bus of Outflow
- **nodes** (*list*) – All oemof.solph components are added to the list

Returns **list** (*Updated list of nodes.*)

dhnx.optimization.add_components.**add_sources**(*on*, *it*, *c*, *labels*, *nodes*, *busd*)

The oemof.solph.Source components for the producers and consumers are initialised based on the given tabular information of the investment_options of the OemofInvestOptimizationModel. Time-series can also be used as attribute values for the outflow of the Source. Therefore, a table with the filename 'source_timeseries' must be given.

Parameters

- **on** (*OemofInvestOptimizationModel*)
- **it** (*DataFrame*) – Table of attributes for Sources for the producers and consumers.
- **c** (*Series*) – Attributes of specific producer or consumer from the ThermalNetwork.
- **labels** (*dict*) – Dictonary containing specifications for label-tuple.
- **nodes** (*list*) – All oemof.solph components are added to the list.
- **busd** (*dict*) – All oemof.solph.Bus objects are given by this dictionary.

Returns **list** (*Updated list of nodes.*)

dhnx.optimization.add_components.**add_storage**(*it*, *labels*, *nodes*, *busd*)

Adds oemof.solph.GenericStorage objects to the list of components.

If attribute *invest* is *True*, the investment version of the Storage is created.

Parameters

- **it** (*pd.DataFrame*) – Table of storage attributes of the producers / consumers.
- **labels** (*dict*) – Dictonary containing specifications for label-tuple.
- **nodes** (*list*) – All oemof.solph components are added to the list.
- **busd** (*dict*) – All oemof.solph.Bus objects are given by this dictionary.

Returns **list** (*Updated list of nodes.*)

dhnx.optimization.add_components.**add_transformer**(*it*, *labels*, *nodes*, *busd*)

Adds oemof.solph.Transformer objects to the list of components.

If attribute *invest* is *True*, an *Investment* attribute is added to the outflow of the Transformer.

Parameters

- **it** (*pd.DataFrame*) – Table of transformer attributes of the producers / consumers.
- **labels** (*dict*) – Dictonary containing specifications for label-tuple.
- **nodes** (*list*) – All oemof.solph components are added to the list.
- **busd** (*dict*) – All oemof.solph.Bus objects are given by this dictionary.

Returns **list** (*Updated list of nodes.*)

9.4.4 dhnx.optimization.oemof_heatpipe

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.oemof_heatpipe.**HeatPipeline**(**args*, ***kwargs*)
Bases: oemof.solph.network.transformer.Transformer

A HeatPipeline represent a Pipeline in a district heating system. This is done by a Transformer with a constant energy loss independent of actual power, but dependent on the nominal power and the length parameter. The HeatPipeline is a single-input-single-output transformer. Additionally, conversion factors for in- and output flow can be applied.

Parameters

- **length** (*float*) – Length of HeatPipeline.
- **heat_loss_factor** (*float*) – Heat loss per length unit as fraction of the nominal power. Can also be defined by a series.

Note: This component is experimental. Use it with care. See also `Transformer`.

The following sets, variables, constraints and objective parts are created

- *HeatPipelineBlock* (if no Investment object present)
- *HeatPipelineInvestBlock* (if Investment object present)

Examples

TODO : example

constraint_group()

class dhnx.optimization.oemof_heatpipe.**HeatPipelineBlock**(**args*, ***kwargs*)
Bases: pyomo.core.base.block.SimpleBlock

Block representing a pipeline of a district heating system. *HeatPipeline*

The following constraints are created:

TODO : Check and fix equations!

$$\dot{Q}_{out}(t) = \dot{Q}_{in}(t) \cdot \frac{\eta_{out}}{\eta_{in}} - \dot{Q}_{loss}(t)$$

$$\dot{Q}_{loss}(t) = f_{loss}(t) \cdot l \cdot \dot{Q}_{nominal}$$

The symbols used are defined as follows (with Variables (V) and Parameters (P)):

symbol	attribute	type	explanation
$\dot{Q}_{out}(t)$	`flow[n, o, t]`	V	Heat output
$\dot{Q}_{in}(t)$	`flow[i, n, t]`	V	Heat input
$\dot{Q}_{loss}(t)$	`heat_loss[n, t]`	P	Heat loss of heat pipeline
$\dot{Q}_{nominal}$	`flows[n, o].nominal_value`	P	Nominal capacity of heating pipeline
η_{out}	`conversion_factors[o][t]`	P	Conversion factor of output flow (Heat Output)
η_{in}	`conversion_factors[i][t]`	P	Conversion factor of input flow (Heat Input)
$f_{loss}(t)$	`heat_loss_factor`	P	Specific heat loss factor for pipeline
l	`length`	P	Length of heating pipeline

CONSTRAINT_GROUP = True

class dhnx.optimization.oemof_heatpipe.**HeatPipelineInvestBlock**(**args*, ***kwargs*)

Bases: `pyomo.core.base.block.SimpleBlock`

Block representing a pipeline of a district heating system. `HeatPipeline`

The following constraints are created:

TODO : Check and fix equations!

$$\dot{Q}_{out}(t) = \dot{Q}_{in}(t) \cdot \frac{\eta_{out}}{\eta_{in}} - \dot{Q}_{loss}(t)$$

$$\dot{Q}_{loss}(t) = f_{loss}(t) \cdot l \cdot \dot{Q}_{nominal}$$

The symbols used are defined as follows (with Variables (V) and Parameters (P)):

symbol	attribute	type	explanation
$\dot{Q}_{out}(t)$	`flow[n, o, t]`	V	Heat output
$\dot{Q}_{in}(t)$	`flow[i, n, t]`	V	Heat input
$\dot{Q}_{loss}(t)$	`heat_loss[n, t]`	V	Heat loss of heat pipeline
$\dot{Q}_{nominal}$	`flows[n, o].nominal_value`	V	Nominal capacity of heating pipeline
η_{out}	`conversion_factors[o][t]`	P	Conversion factor of output flow (heat output)
η_{in}	`conversion_factors[i][t]`	P	Conversion factor of input flow (heat input)
$f_{loss}(t)$	`heat_loss_factor`	P	Specific heat loss factor for pipeline
l	`length`	P	Length of heating pipeline

CONSTRAINT_GROUP = True

class dhnx.optimization.oemof_heatpipe.**Label**
Bases: dhnx.optimization.oemof_heatpipe.solph_label

9.4.5 dhnx.optimization.precalc_hydraulic

This module is designed to hold functions for pre-calculation of hydraulic parameters for the district heating network dimensioning.

The aim is to calculate the maximum heat transport capacity of pipelines given a maximum pressure drop per meter, roughness of the pipes inner surface, and an estimated delta T of the forward and return pipes.

The equations and values used for the calculation can be found here: http://www.math-tech.at/Beispiele/upload/gra_Druckverlust_in_Rohrleitungen.PDF https://www.schweizer-fn.de/stroemung/rauhigkeit/rauhigkeit.php

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

dhnx.optimization.precalc_hydraulic.**calc_Re**(*v*, *d_i*, *k_v*)
Calculates the Reynolds number for a given velocity, inner diameter and kinematic viscosity

$\frac{v \cdot d_i}{\nu}$

Parameters

- **v** (*numeric*) – v: flow velocity [m/s]
- **d_i** (*numeric*) – d_i: inner pipe diameter [m]
- **k_v** (*numeric*) – ν: kinematic viscosity [m^2/s]

Returns **Reynolds number [-]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_d_p**(*lam*, *length*, *d_i*, *d*, *v*)
Calculates the pressure drop in a pipe for a given Darcy friction factor

$\Delta p = \lambda \frac{l}{d_i} \frac{\rho}{2} v^2$

Parameters

- **lam** (*numeric*) – λ: Darcy friction factor [-]
- **length** (*numeric*) – l: length of the pipe [m]
- **d_i** (*numeric*) – d_i: inner pipe diameter [m]
- **d** (*numeric*) – ρ: density [kg/m^3]
- **v** (*numeric*) – v: flow velocity [m/s]

Returns **Pressure drop [Pa]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_k_v**(*d_v*, *d*)
Calcutlates the kinematic viscosity for given density and dynamic viscosity

$\nu = \frac{\eta}{\rho}$

Parameters

- **d_v** (*numeric*) – η: dynamic viskosity [kg/(m*s)]
- **d** (*numeric*) – ρ: density [kg/m^3]

Returns **kinematic viscosity [m^2/s]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_lambda_laminar**(*Re*)

Calculates the Darcy friction factor for a given Reynolds number for a laminar flow

$\lambda = \frac{64}{Re}$

Parameters **Re** (*numeric*) – Re: Reynolds number [-]

Returns **Darcy friction factor [-]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_lambda_rough**(*d_i*, *k*)

Calculates the Darcy friction factor for a turbulent flow and a rough inner pipe surface (Prandtl & Nikuradse)

$\lambda = \frac{1}{(2log(3{,}71\frac{d_i}{k}))^2}$

Parameters

- **d_i** (*numeric*) – d_i: inner pipe diameter [m]
- **k** (*numeric*) – k: roughness of inner pipeline surface [mm]

Returns **Darcy friction factor [-]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_lambda_transition**(*R_e*, *k*, *d_i*)

Calculates the Darcy friction factor for a given Reynolds number for a turbulent flow and the transition area between a rough and smooth pipe surface.

See also `eq_transition()`.

Parameters

- **R_e** (*numeric*) – Re: Reynolds number [-]
- **k** (*numeric*) – k: roughness of inner pipeline surface [mm]
- **d_i** (*numeric*) – d_i: inner pipe diameter [m]

Returns **Darcy friction factor [-]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_lambda_turb1**(*Re*)

Calculates the Darcy friction factor for a given Reynolds number for a turbulent flow, a smooth pipe and a Reynolds number smaller than 10^5 (Blasius)

$\lambda = 0,3164 \cdot Re^{-0,25}$

Parameters **Re** (*numeric*) – Re: Reynolds number [-]

Returns **Darcy friction factor [-]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_lambda_turb2**(*Re*)

Calculates the Darcy friction factor for a given Reynolds number for a turbulent flow, a smooth pipe and a Reynolds number between 10^5 and 10^6 (Nikuradse)

$\lambda = 0,0032 + 0,221 \cdot Re^{-0,237}$

Parameters **Re** (*numeric*) – Re: Reynolds number [-]

Returns **Darcy friction factor [-]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_lambda_turb3**(*Re*)

Calculates the Darcy friction factor for a given Reynolds number for a turbulent flow, a smooth pipe and a Reynolds number higher than 10^6. For a formula, see `eq_smooth()`.

Parameters **Re** (*numeric*) – Re: Reynolds number [-]

Returns **Darcy friction factor [-]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_mass_flow**(*v*, *di*, *T_av*, *p=101325*)

Calculates the mass flow in a pipe for a given density, diameter and flow velocity. The average temperature is needed for a correct value of the density.

$\dot{m} = \pi \rho_{T_a v} v \left(\frac{d_i}{2}\right)^2$

Parameters

- **v** (*numeric*) – v: flow velocity [m/s]
- **di** (*numeric*) – d_i: inner diameter [m]
- **T_av** (*numeric*) – $T_a v$: temperature level [°C]
- **p** (*numeric*) – p: pressure [Pa]

Returns **mass flow [kg/s]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_mass_flow_P**(*P*, *T_av*, *delta_T*, *p=101325*)

Calculates the mass flow in a pipe for a given power and heat difference. The average temperature is needed for a correct value of the heat capacity.

$\dot{m} = \frac{P}{c_{P_{T_{av}}} \cdot \Delta T}$

Parameters

- **P** (*numeric*) – P: power [W]
- **T_av** (*numeric*) – T_{av}: average temperature [°C]
- **delta_T** (*numeric*) – ΔT: temperature difference [K]
- **p** (*numeric*) – p: pressure [Pa]

Returns **mass flow [kg/s]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_pipe_loss**(*temp_average*, *u_value*, *temp_ground=10*)

Calculates the heat loss of a DHS pipe trench.

Temperatures must be given in the same unit, K or °C.

$P_{loss} = (T_{average} - T_{ground}) \cdot U$

Parameters

- **temp_average** (*float*) – $T_{average}$: Average temperature of medium in (if u_value relates to forward and return pipe, the average temperature of forward and return must be given.)
- **u_value** (*float*) – U: Heat transmission coefficient of whole trench in W/(m*K) (u_value of forward and return pipe must be summed up, if total heat loss should be calculated.)
- **temp_ground** (*float*) – T_{ground}: Temperature of surrounding, e.g. ground.

Returns **Heat loss of pipe trench [W/m]** (*float*)

dhnx.optimization.precalc_hydraulic.**calc_power**(*T_vl=80*, *T_rl=50*, *mf=3*, *p=101325*)

Function to calculate the thermal power based on mass flow and temperature difference.

$P_{th} = \dot{m} \cdot (c_{p_{VL}} \cdot T_{VL} - c_{p_{RL}} \cdot T_{RL})$

Parameters

- **T_vl** (*numeric*) – T_{VL}: forward temperature [°C]
- **T_rl** (*numeric*) – T_{RL}: return temperature [C°]
- **mf** (*numeric*) – $\dot{m}$: mass flow [kg/s]

- **p** (*numeric*) – p: pressure [Pa]

Returns **thermal power [W]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_v**(*vol_flow*, *d_i*)

Calculates the velocity for a given volume flow and inner diameter of a pipe.

$v_{flow} = \frac{\dot{V}}{(\frac{d_i}{2})^2 * \pi}$

Parameters

- **vol_flow** (*numeric*) – $\dot{V}$: volume flow [m³/h]
- **d_i** (*numeric*) – d_i: inner diameter [m]

Returns **flow velocity [m/s]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**calc_v_mf**(*mf*, *di*, *T_av*, *p=101325*)

Calculates the flow velocity for a given mass flow and inner diameter. The average temperature is needed for a correct value of the density.

$v = \frac{\dot{m}}{\pi \rho \cdot \left(\frac{d_i}{2}\right)^2}$

Parameters

- **mf** (*numeric*) – $dotm$: mass flow [kg/s]
- **di** (*numeric*) – d_i: inner diameter [m]
- **T_av** (*numeric*) – T_{av}: average temperature [°C]

  p: numeric p: pressure [Pa]

Returns **flow velocity [m/s]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**delta_p**(*v*, *d_i*, *k=0.1*, *T_medium=90*, *length=1*, *pressure=101325*, *R_crit=2320*, *fluid='IF97::Water'*)

Function to calculate the pressure loss in a pipeline

Parameters

- **v** (*numeric*) – v: flow velocity [m/s]
- **d_i** (*numeric*) – d_i: inner pipe diameter [m]
- **k** (*numeric*) – k: roughness of inner pipeline surface [mm]
- **T_medium** (*numeric*) – T_{medium}: fluid temperature [°C]
- **length** (*numeric*) – l: length of the pipe [m]
- **pressure** (*numeric*) – p: pressure in the pipe [Pa]
- **R_crit** (*numeric*) – Re_{crit}: critical Reynolds number between laminar and turbulent flow [-]
- **fluid** (*str*) – name of the fluid used

Returns **Pressure drop [bar]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**eq_smooth**(*x*, *R_e*)

Calculation of the pressure drop of hydraulic smooth surfaces. (Prandtl & Karman)

$f(x) = x - 2 \cdot log\left(\frac{Re}{2{,}51x}\right)$

Parameters

- **x** (*numeric*) – x: function variable [-]
- **R_e** (*numeric*) – Re: Reynolds number [-]

Returns **Equation** (*numeric*)

dhnx.optimization.precalc_hydraulic.**eq_transition**(*x*, *R_e*, *k*, *d_i*)

Equation to be solved for the transition range between a smooth and rough pipe surface (Prandtl-Colebrook)

$f(x) = x + 2 \cdot log\left(\frac{2{,}51x}{Re} \cdot \frac{k}{3{,}71d_i}\right)$

Parameters

- **x** (*numeric*) – x: function variable () [-]
- **R_e** (*numeric*) – Re: Reynolds number [-]
- **k** (*numeric*) – k: roughness of inner pipeline surface [mm]
- **d_i** (*numeric*) – d_i: inner diameter [m]

Returns **Equation** (*numeric*)

dhnx.optimization.precalc_hydraulic.**v_max_bisection**(*d_i*, *T_average*, *k=0.1*, *p_max=100*, *p_epsilon=0.1*, *v_epsilon=0.001*, *v_0=0.01*, *v_1=10*, *pressure=101325*, *fluid='IF97::Water'*)

Calculates the maximum velocity via bisection for a given pressure drop.

The two starting values *v_0* and *v_1* need to be given, with *v_0* below the expected flow velocity and *v_1* above. These are the starting values for the bi-section method.

If either of the stop-criteria *p_epsilon* or *v_epsilon* is reached, the iterative calculation is stopped.

Parameters

- **d_i** (*numeric*) – d_i: inner diameter [m]
- **T_average** (*numeric*) – T_{av}: average temperature [°C]
- **k** (*numeric*) – k: roughness of inner pipeline surface [mm]
- **p_max** (*numeric*) – p_{max}: maximum pressure drop in pipeline [Pa]
- **p_epsilon** (*numeric*) – p_ϵ: accuracy of pressure [Pa]
- **v_epsilon** (*numeric*) – v_ϵ: accuracy of velocity [m/s]
- **v_0** (*numeric*) – v_0: first value of initial guess for maximum flow velocity [m/s]
- **v_1** (*numeric*) – v_1: second value of initial guess for maximum flow velocity [m/s]
- **pressure** (*numeric*) – p: pressure level [Pa]
- **fluid** (*str*) – type of fluid, default: 'IF97::Water'

Returns **maximum flow velocity [m/s]** (*numeric*)

dhnx.optimization.precalc_hydraulic.**v_max_secant**(*d_i*, *T_average*, *k=0.1*, *p_max=100*, *p_epsilon=1*, *v_0=1*, *v_1=2*, *pressure=101325*, *fluid='IF97::Water'*)

Calculates the maximum velocity via iterative approach using the secant method.

The two different starting values v_0 and v_1 should be in the area of the maximum flow velocity, as iteration starts from there.

Parameters

- **d_i** (*numeric*) – d_i: inner diameter [m]
- **T_average** (*numeric*) – T_{av}: average temperature [°C]
- **k** (*numeric*) – k: roughness of inner pipeline surface [mm]
- **p_max** (*numeric*) – p_{max}: maximum pressure drop in pipeline [Pa]
- **p_epsilon** (*numeric*) – p_ϵ: accuracy of pressure [Pa]
- **v_0** (*numeric*) – v_0: first value of initial guess for maximum flow velocity [m/s]
- **v_1** (*numeric*) – v_1: second value of initial guess for maximum flow velocity [m/s]
- **pressure** (*numeric*) – p: pressure level [pa]
- **fluid** (*str*) – type of fluid, default: 'IF97::Water'

Returns **maximum flow velocity [m/s]** (*numeric*)

9.5 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** (*pd.DataFrame*) – DataFrame containing the sequences

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

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

Returns **concat_sequences** (*pd.DataFrame*) – DataFrame containing the sequences

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** (*pd.DataFrame*) – DataFrame with all know mass flow of consumers and producer.

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

Returns **pipes_mass_flow** (*pd.DataFrame*) – Mass flow in the pipes [kg/s]

_calculate_reynolds()
Calculates the Reynolds number.

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

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

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

_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** (*pd.DataFrame*) – DataFrame with distributed pressure losses for inlet and return for every time step and pipe [Pa]

_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** (*pd.DataFrame*) – Localized pressure losses at the nodes [Pa]

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

_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** (*pd.Series*) – Pump power [W]

_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** (*np.matrix*) – Constant part of the exponent [kg/s]

_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** (*pd.DataFrame*) – DataFrame containing temperatures for all nodes [°C]

_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** (*pd.DataFrame*) – Return temperature with the consumers values set [°C]

_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** (*pd.DataFrame*) – Heat losses in the pipes [W]

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

Parameters **thermal_network**

Returns **results** (*dict*)

CHAPTER 10

What's New

These are new features and improvements of note in each release

Releases

10.1 v0.0.3 (???)

10.1.1 API changes

- something

10.1.2 New features

- Pre-calculation function for optimisation parameters

Geometry processing:

- New method for connecting buildings to polygons boundary
- Allow to keep index of buildings/consumer

10.1.3 New components/constraints

- something

10.1.4 Documentation

- Fix/add complete API documentation of gistools and optimization modules

10.1.5 Bug fixes

- something

10.1.6 Known issues

- something

10.1.7 Testing

- something

10.1.8 Other changes

- Refactor internal structure of optimization modules
- Improve processing of optimisation results by returning raw (un-rounded) solver results for each pipeline
- Use tox for CI automation
- Rename examples folder of optimisation examples

10.1.9 Contributors

- Johannes Röder
- Joris Zimmermann
- Robert Valeske

10.2 v0.0.2 (09.07.2021)

10.2.1 API changes

- nothing new

10.2.2 New features

- Geometry processing modules (`dhnx.gistools`) for creating the ThermalNetwork structure (pipes, forks, consumers, producers) from user-defined geo-referenced input data. This could also be an import from osmnx.

10.2.3 New components/constraints

- nothing new

10.2.4 Documentation

- nothing new

10.2.5 Bug fixes

- Avoid bidirectional flow at consumers connections in any case.
- Fixed error of simulation being dependent on the definition of the direction of an edge. When swapping the direction of an edge in a simple example, wrong results were the consequence. This has been fixed such that the simulation does not depend on the edge direction, but the actual mass flow.

10.2.6 Known issues

- Simulation: Calculations rely on defined pipe direction
- OSMNetworkImporter not properly working

10.2.7 Testing

- nothing new

10.2.8 Other changes

- nothing new

10.2.9 Contributors

- Johannes Röder
- Joris Zimmermann
- Uwe Krien

10.3 v0.0.1 (28.10.2020)

First release by the oemof developing group.

10.3.1 Contributors

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

CHAPTER 11

Authors

–**alphabetic order**–

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

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

C

D

E

F

G

H

I