hynet Documentation

Release 1.2.3

Matthias Hotz

May 02, 2021

Contents

CHAPTER 1

Welcome to *hynet*

hynet is a package for the computation of the optimal power flow (OPF) in hybrid AC/DC power systems, i.e., the cost- or loss-minimizing allocation of generation resources and the corresponding system state to serve a given load while satisfying the system's technical boundary conditions. *hynet* supports power systems that comprise an arbitrary interconnection of AC grids and radial DC grids, i.e., point-to-point and radial multi-terminal HVDC systems. With respect to OPF methods, it supports the solution of the nonconvex quadratically constrained quadratic program (QCQP) as well as its semidefinite relaxation (SDR) and second-order cone relaxation (SOCR). For more information, please refer to *hynet*'s documentation (HTML/PDF) for a description the software and this publication (preprint) or this dissertation for the mathematical background. *hynet* uses SQLite-based SQL databases to store grid infrastructure and scenario information. A library with several grid databases is provided here.

1.1 Installation

hynet was developed for Python 3.5 and higher and requires NumPy, SciPy, pandas, SQLAlchemy, Matplotlib, tqdm, h5py as well as at least one of the supported *solvers*. For a convenient installation, the Python distribution Anaconda (or the stripped-down Miniconda) may be used, where the included package manager Conda supports a straightforward installation of the supported solvers.

To install *hynet* using Python's package management system, run

```
pip install hynet
```

The installation of *hynet* and the installed *solvers* can be tested with

```
python -m hynet test
```

To install *hynet* from its sources, get the latest source code by cloning the *hynet* repository with Git via

```
git clone https://gitlab.com/tum-msv/hynet.git
```

and initiate the installation with

```
python setup.py install
```

1.1.1 Solvers

In the following, the supported solvers are listed. **Currently, the utilization of the following solvers is recommended: IPOPT for the QCQP, MOSEK for the SDR, and MOSEK or CPLEX for the SOCR.** Regarding the latter, it was found empirically that CPLEX is more robust while MOSEK is computationally more efficient. Please note that **even if only QCQPs are solved, it is recommended to install MOSEK or CPLEX**, as they enable the efficient computation of an initial point for QCQP solvers.

1.1.1.1 IPOPT

IPOPT is an open-source software package for large-scale nonlinear optimization and CYIPOPT is a Python wrapper for IPOPT. With Conda, both can be installed as follows.

- Linux and MAC OS X:

```
conda install -c conda-forge cyipopt
```

- Windows:

```
conda install -c pycalphad cyipopt
```

1.1.1.2 MOSEK

MOSEK is an interior-point optimizer for large-scale conic optimization problems. It is commercial, but offers a free academic license. With Conda, MOSEK can be installed with

```
conda install -c mosek mosek
```

hynet's SDR solver interface for MOSEK supports a chordal conversion of the semidefinite program to enable the computation of the SDR for medium- and large-scale systems with a viable computational effort. To utilize the chordal SDR, CHOMPACK, a library for chordal matrix computations, and CVXOPT, a Python package for convex optimization, are required. With Python's package management system, both can be installed with

```
pip install chompack cvxopt
```

1.1.1.3 IBM ILOG CPLEX

CPLEX is a high-performance mathematical programming solver for linear, mixed integer, quadratic, and quadratically constrained programming problems. It is commercial, but offers a free academic license through the IBM Academic Initiative. For the installation, please refer to the instructions provided with CPLEX as well as the section "Setting up the Python API of CPLEX" of the CPLEX documentation.

1.1.1.4 PICOS

hynet supports the solution of the SDR and SOCR with PICOS. However, the additional modeling layer causes a performance drawback. PICOS is an open-source Python-based modeling language for linear and conic optimization problems. It supports several solvers, including the open-source solver CVXOPT. With Python's package management system, PICOS and CVXOPT can be installed with

```
pip install picos cvxopt
```

1.1.1.5 PYOMO

hynet supports the solution of the QCQP with Pyomo. However, the additional modeling layer causes a performance drawback. Furthermore, the import of Pyomo is demanding and **slows down the import** of *hynet* significantly, thus the installation is only recommended if Pyomo is actually utilized. Pyomo is an open-source optimization modeling language and includes support for the solver IPOPT. With Conda, both can be installed with

```
conda install -c conda-forge pyomo libgfortran
conda install -c cachemeorg ipopt_bin
```

1.2 Usage

Open a terminal, navigate to the directory that contains the grid databases, and start a Python shell, either the standard shell (`python`) or a more convenient one like IPython or ptpython. At the Python command prompt, import *hynet* via

```
import hynet as ht
```

To access the data of the system in the file `pjm_hybrid.db`, connect to this database using

```
database = ht.connect('pjm_hybrid.db')
```

The optimal power flow for the default scenario of this system can then be calculated with

```
result = ht.calc_opf(database)
```

The object `result` contains all result data. For example, to print a summary, print details of the solution, and access the determined bus voltages, type

```
print(result)
print(result.details)
result.bus['v']
```

By default, *hynet* selects the most appropriate QCQP solver among those installed. To specify the type of solver explicitly, set the `solver_type` as illustrated below.

```
ht.calc_opf(database, solver_type=ht.SolverType.QCQP)
ht.calc_opf(database, solver_type=ht.SolverType.SDR)
ht.calc_opf(database, solver_type=ht.SolverType.SOCR)
```

In case that the scenario shall be modified prior to the OPF calculation, it can be loaded explicitly via

```
scenario = ht.load_scenario(database)
```

For example, to set the load at bus 2 to 100MW and 50Mvar, use

```
scenario.bus.at[2, 'load'] = 100 + 50j
```

The optimal power flow for this modified scenario can be calculated with

```
ht.calc_opf(scenario)
```

For more information and usage examples, please refer to the tutorials in USAGE.md, *hynet*'s documentation (HTML/PDF), and this publication (preprint).

1.3 Contributing

Contributions to *hynet* are very welcome. Please refer to CONTRIBUTING.md for more information. In case that *hynet* is useful to you, we would appreciate if you star this project.

1.4 Credits

This software was developed at the Professur für Methoden der Signalverarbeitung, Technische Universität München (TUM). The principal developer and project maintainer is Matthias Hotz (@matthias_hotz), who would like to recognize the highly appreciated support of the following contributors:

- Vincent Bode (TUM): Database management, network graph export
- Michael Mitterer (TUM): Distributed computation, MATPOWER import, database management
- Christian Wahl (TUM): Capability region visualizer, CI configuration
- Yangyang He (TUM): CVXPY and PICOS solver interface
- Julia Sistermanns (TUM): Feature- and structure-preserving network reduction

1.5 Citation

In case that *hynet* is used in the preparation of a scientific publication, we would appreciate the citation of the following work:

M. Hotz and W. Utschick, "*hynet:* An Optimal Power Flow Framework for Hybrid AC/DC Power Systems," IEEE Transactions on Power Systems, vol. 35, no. 2, pp. 1036-1047, Mar. 2020.

The corresponding BibTeX entry is provided below.

```
@article{Hotz2020,
    Author = {Matthias Hotz and Wolfgang Utschick},
    Journal = {IEEE Transactions on Power Systems},
    Title = {\textit{{hynet}:} {A}n Optimal Power Flow Framework for Hybrid {AC}/
↪{DC} Power Systems},
    Year = {2020},
    Month = {March},
    Volume = {35},
    Number = {2},
    Pages = {1036-1047},
    Doi = {10.1109/TPWRS.2019.2942988}}
```

Furthermore, in case that the feature- and structure-preserving network reduction functionality in *hynet* is utilized, we would appreciate the citation of the following work:

J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, "Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids," 13th IEEE PES PowerTech Conf., Milano, Italy, Jun. 2019.

1.6 License

CHAPTER 2

Tutorial

In the following, some use cases are presented to illustrate the application of *hynet*. In all subsections, it is assumed that a Python shell was started in a directory that contains the grid databases and that the following commands were executed:

```
import hynet as ht
database = ht.connect('pjm_hybrid.db')
```

2.1 Optimal Power Flow

2.1.1 Selection and Configuration of a Specific Solver

The README.md illustrates the automatic selection of a QCQP solver and the selection of a solver for a specific OPF formulation (QCQP, SDR, SOCR). However, in certain cases, e.g. for reproducibility, the explicit selection of a specific solver with specific parameters may be desired. To this end, a list of solver classes for the available solvers is provided by

```
ht.AVAILABLE_SOLVERS
```

For example, to solve the OPF problem with IPOPT, use

```
solver = ht.solver.ipopt.QCQPSolver()
result = ht.calc_opf(database, solver=solver)
print(result)
```

To change the convergence tolerance of IPOPT to 10^{-7}, extend the `param` dictionary accordingly, i.e.,

```
solver.param['tol'] = 1e-7
print(ht.calc_opf(database, solver=solver))
```

For more information on the parameters of a solver class, please refer to its documentation, e.g. via

```
help(ht.solver.ipopt.QCQPSolver)
```

2.1.2 Analysis of the Optimal Power Flow Result

The result of an OPF calculation contains extensive information about the resulting system state, the solution process, as well as the associated optimization problem. An overview of the available data is provided the result object's help, i.e.,

```
result = ht.calc_opf(database)
help(result)
```

A formatted summary can be printed with

```
print(result)
```

while detailed information related to the buses, branches, converters, and injectors can be pretty-printed using

```
print(result.details)
```

The data related to these entities is stored in pandas data frames, which provide extensive support for data analysis. For example, the total active power injection (in MW) and the three buses with the smallest voltage magnitude can be determined with

```
result.injector['s'].sum().real
result.bus['v'].abs().nsmallest(n=3)
```

For the visual inspection of the data, Matplotlib may be used, where *hynet* includes some functions for some common plotting tasks, i.e.,

```
ht.show_voltage_profile(result)
ht.show_dispatch_profile(result)
ht.show_branch_flow_profile(result)
ht.show_converter_flow_profile(result)
```

Furthermore, in case of exactness of the SDR or SOCR relaxation, the dual variables of the power balance constraints equal the locational marginal prices (LMPs), which can be illustrated with

```
ht.show_lmp_profile(result)
```

Finally, to identify congested branches in the system, the dual variables of the ampacity constraint may be investigated via

```
ht.show_ampacity_dual_profile(result, id_label=True)
```

When processing the result programmatically, it is important to check its validity:

```
result.is_valid
```

This check ensures the following conditions:

1) The result contains solution data:

```
```python
not result.empty
```
```

2) The solver reported the successful solution of the problem:

```
```python
result.solver_status == ht.SolverStatus.SOLVED
```
```

3) The solution complies with the tolerances for a physical solution:

```
```python
result.is_physical
```

The solution is considered physically valid if the power balance and converter flow error complies with the respective tolerances, which can be verified individually by

```python
result.has_valid_power_balance, result.has_valid_converter_flows
```
```

The third item is particularly important if the OPF problem was solved via a relaxation (SDR or SOCR), where inexactness of the relaxation can lead to a nonphysical solution. In some cases, it can be appropriate to consider the above conditions individually, e.g., to check only the first and third item to accept also solutions for which the solver obtained a result but reported numerical issues.

2.1.3 Optimal Power Flow for Different Scenarios

The *hynet* grid database format comprises two major categories of data, i.e., a description of the *grid infrastructure* as well as *scenarios*. The grid infrastructure specifies all physical entities of the grid, i.e., buses, lines, transformers, converters, shunts, and injections, where the latter encompasses conventional as well as renewables-based generation, dispatchable loads, and prosumers. This physical description is complemented by scenario information, which specifies the load, injector capabilities (e.g., for renewable energy sources), and inactivity of certain entities (e.g. decommitment of generators) at particular time instants. To inspect the scenarios provided by the database, type

```
scenario_info = ht.get_scenario_info(database)
print(scenario_info)
```

For example, assume we want to calculate the OPF for all scenarios of the exemplary winter weekday in this database. We start by extracting these scenarios and sort them by their time stamp:

```
scenario_info = scenario_info.loc[scenario_info['name'] == 'Winter Weekday']
scenario_info.sort_values(by='time', inplace=True)
```

Now, we calculate the OPF for all these scenarios and store the minimum injection cost in a new column of the data frame:

```
for id_ in scenario_info.index:
    result = ht.calc_opf(database, scenario_id=id_)
    scenario_info.loc[id_, 'injection_cost'] = result.get_total_injection_cost()
```

Finally, we visualize the cost using Matplotlib:

```
import matplotlib.pyplot as plt
plt.plot(scenario_info['time'], scenario_info['injection_cost']/1e3)
plt.xlabel('Hour of the Day')
plt.ylabel('Cost in k$')
plt.title('Injection Cost for an Exemplary Winter Weekday')
plt.show()
```

2.1.4 Distributed Computation of Optimal Power Flows

The calculation of several OPFs, e.g., to analyze different scenarios of a grid, can be computationally demanding. To this end, *hynet* includes support for a distributed solution on a server cluster. Consider again the analysis of the examplary winter weekday, where the hourly injection cost was computed in *Optimal Power Flow for Different*

Scenarios. In order to distribute the calculation of these OPFs to several servers, start a *hynet* optimization server via

```
server = ht.start_optimization_server()
```

To see more options for the server, call `help(ht.start_optimization_server)`. After starting the server, connect *hynet* optimization clients by logging in to the **client machines** and, in the terminal, run

```
python -m hynet client [server_ip]
```

where `[server_ip]` is the network IP address of the machine running the *hynet* optimization server. To see more options for the clients, call `python -m hynet client -h`. Then, back at the Python prompt on the **server machine**, load the scenarios of the exemplary winter weekday into a list using

```
scenario_info = ht.get_scenario_info(database)
scenario_info = scenario_info.loc[scenario_info['name'] == 'Winter Weekday']
scenario_info.sort_values(by='time', inplace=True)
scenarios = [ht.load_scenario(database, scenario_id=id_) for id_ in scenario_info.
↪index]
```

and distribute their computation to the client machines using

```
results = server.calc_jobs(scenarios)
```

This call is blocking until all jobs are processed. The obtained OPF results are returned in `results`, corresponding to the order in `scenarios`. Finally, the *hynet* optimization server and all clients are shut down via

```
server.shutdown()
```

Remarks:

- If the client machines are unavailable, the optimization server can be operated in a local mode, see the parameter `local` of `ht.start_optimization_server`.
- To automate the start of *hynet* optimization clients, the method `server.start_clients` may be utilized. For example, starting *hynet* optimization clients on the machines with the host names `client1` and `client2` via SSH using the user name `my_user_name` (with automated authentication via SSH keys), where the *hynet* optimization server is running on the machine with the IP address `10.0.0.5`, can be achieved with

```
server.start_clients(['client1', 'client2'], '10.0.0.5',
                     ssh_user='my_user_name', num_workers=4,
                     log_file='hynet_client.log')
```

In this configuration, every *hynet* optimization client offers 4 worker processes and logs its output to `hynet_client.log`.

2.1.5 Optimal Power Flow for a Customized Scenario

A scenario may also serve as the starting point for a further analysis of the grid, e.g., to study the impact of outages. In such cases, the desired scenario can be loaded explicity. For example, the base case is loaded with

```
scenario = ht.load_scenario(database)
```

Let us first inspect the data of this scenario with

```
print(scenario)
```

To get more information on the type of data, we can consult the class documentation, i.e.,

```
help(scenario)
```

Now, assume that we want to study the impact of an outage of the AC/DC converter between bus 5 and 8 on the optimal power flow. To this end, we remove the respective converter with ID 3 from the scenario and initiate an OPF calculation:

```
scenario.converter.drop(3, inplace=True)
print(ht.calc_opf(scenario))
```

As another example, let us study the hour 6pm to 7pm of the exemplary winter weekday with scenario ID 67, see also *Optimal Power Flow for different Scenarios*.

```
scenario = ht.load_scenario(database, scenario_id=67)
```

Assume we want to take bus 3 off the grid, including all entities connected to it, and study the OPF. As bus 3 is the reference bus, we move the reference to bus 5. These actions can be implemented by

```
scenario.remove_buses([3])
scenario.bus.loc[5, 'ref'] = True
print(ht.calc_opf(scenario))
```

2.1.6 Optimal Power Flow with Loss Minimization

Typically, the OPF is considered with respect to minimum injection cost, but sometimes an OPF with minimum transmission losses is more appropriate - from an engineering as well as mathematical perspective. To illustrate this, consider again the default scenario, i.e.,

```
scenario = ht.load_scenario(database)
```

This system exhibits the hybrid architecture and satisfies the conditions for the related results on exactness of the SDR and SOCR, which is verified by

```
scenario.verify_hybrid_architecture_conditions()
```

Thus, the use of an SOCR solver (which we assume to be available on your system) is appropriate:

```
result = ht.calc_opf(scenario, solver_type=ht.SolverType.SOCR)
print(result)
```

The theory guarantees exactness of the relaxation as long as no *pathological price profile* occurs, which is indeed the case:

```
result.is_physical, result.reconstruction_mse
```

Now, assume all generation in the system is based on renewable energy sources, which are often considered with zero marginal costs. Correspondingly, we set all cost functions to zero and calculate the OPF again, i.e.,

```
scenario.injector['cost_p'] = None
result = ht.calc_opf(scenario, solver_type=ht.SolverType.SOCR)
```

Pathological price profiles are characterized by a union of linear subspaces in the domain of the dual variables and, as they intersect at the origin, inexactness of the relaxation is more likely if the LMP is (close to) zero at some buses. Indeed, although this system features the hybrid architecture, the relaxation has now become inexact:

```
print(result)
result.is_physical, result.reconstruction_mse
```

The observation of a pathological price profile can motivate a reconsideration from an engineering perspective: If the LMP for active power is zero at some buses, those locations provide power free of charge and, there, the OPF's

cost minimization objective remains without effect. In such cases, it is reasonable to consider transmission losses in the objective to avoid the waste of freely available energy. In *hynet*, the OPF's objective can be augmented with a *loss penalty term* by imposing an (artificial) cost on the electrical losses. For example, this cost is set to $1/MWh using

```
scenario.loss_price = 1
```

While this penalty term is motivated from an engineering perspective, it also establishes exactness of the relaxation:

```
result = ht.calc_opf(scenario, solver_type=ht.SolverType.SOCR)
print(result)
result.is_physical, result.reconstruction_mse
```

The recovery of exactness is explained by the impact of the loss term, which can be shown to introduce an offset to the marginal prices of the injectors and, therewith, induces a shift of the price profile that potentially avoids the critical subspaces.

2.1.7 Saving an Optimal Power Flow Result

As OPF results can be reproduced rather easily and as it is often not necessary to store the entirety of an OPF result, the *hynet* grid database format favors simplicity and does not support the storage of results. However, if the necessity of storing an OPF result arises, e.g., to avoid its repeated computation, it can be serialized with pickle. Let us calculate an OPF result and store it to the file `my_result.pickle`:

```
import pickle
result = ht.calc_opf(database)
with open('my_result.pickle', 'wb') as file:
    pickle.dump(result, file)
```

We can load the result again using

```
with open('my_result.pickle', 'rb') as file:
    result = pickle.load(file)
```

Caution: Note that pickles are not secure against malicious modifications and should only be opened if they were received from a trusted source.

2.1.8 Tracking the Solution Progress

By default, *hynet* does not show any progress information to avoid cluttering the standard output. However, especially for large grids that involve a significant computation time, such information may be desired to track the solution progress. To this end, the progress within the solver can be tracked by enabling its verbose mode, e.g.,

```
solver = ht.solver.ipopt.QCQPSolver(verbose=True)
print(ht.calc_opf(database, solver=solver))
```

while the progress within *hynet* may be tracked by enabling its debug log output, i.e.,

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

2.2 Additional Problem Formulations

The object-oriented design of *hynet* facilitates extensions to OPF-related problem formulations. These extensions consist of an adapted mapping from the scenario data to the QCQP formulation as well as an adjusted result representation, while sharing the other software infrastructure. Correspondingly, many aspects of *hynet* that are

documented in the context of OPF problems apply to extensions as well. In the following, the extensions that are included with *hynet* are briefly presented, while further information can be found in the package documentation.

2.2.1 Maximum Loadability

The *maximum loadability* is an important characteristic of a power system and, e.g., relevant in expansion planning and voltage stability assessment. In *hynet*, the maximum loadability problem as proposed in Section 3 of "Maximum loadability of power systems using interior point nonlinear optimization method" by Irisarri et al. is included, i.e., the power balance equations are extended with a scaled load increment and the scaling of the increment is maximized. The nodal load increment is defined by the column `'load_increment'` in the `bus` data frame of the scenario and, if left unspecified, it is set to the nodal load (i.e., a constant power factor is maintained).

The maximum loadability for the default scenario of the grid database can be calculated with

```
result = ht.calc_loadability(database)
print(result)
```

The load increment is set to the system's load, which is verified with

```
result.scenario.bus['load_increment']
```

and the maximum load increment scaling can be accessed via

```
result.load_increment_scaling
```

A custom load increment, e.g., only at bus 2 and 3, can be configured as follows. Load the scenario,

```
scenario = ht.load_scenario(database)
```

set the load increment accordingly,

```
scenario.bus['load_increment'] = 0.0
scenario.bus.loc[2:3, 'load_increment'] = scenario.bus.loc[2:3, 'load']
```

and initiate the computation of the maximum loadability,

```
result = ht.calc_loadability(scenario)
print(result)
```

The identified maximum load is thus given by

```
scenario.bus['load'] + result.load_increment_scaling * scenario.bus['load_increment
↪']
```

which is also available via the result, which contains a copy of the original scenario with the maximum load:

```
result.scenario.bus['load']
```

2.3 Management of Grid Databases

The *hynet* grid database format divides the data into two categories, the *grid infrastructure* as well as *scenario information* (see also *Optimal Power Flow for Different Scenarios*). The data considered as grid infrastructure comprises the system's physical entities as well as their parameters that are assumed to be fixed for the purpose of OPF studies, while scenario information comprises certain OPF-relevant time-dependent parameters. In particular, the latter can capture (with `scenario` being a scenario object):

- Changes of the load (`scenario.bus['load']`).
- Selected changes of injectors:

  - Scaling of the cost function for active and reactive power (`scenario.injector[['cost_p', 'cost_q']]`). The scaling must be the same for both cost functions of an injector.
  - Changes of the box constraints of the capability region (`p_min`, `p_max`, `q_min`, and `q_max` of the objects in `scenario.injector['cap']`).
- Inactivity of injectors, branches, converters, or buses (captured by dropping the respective rows in `scenario.injector`, `scenario.branch`, `scenario.converter`, and `scenario.bus`).
- Inactivity of shunt compensation (captured by setting the respective entry in `scenario.bus['y_tld']` to zero).

In the following, it is illustrated how grid infrastructure and scenario data can be saved in *hynet* grid databases, while the subsequent sections discuss further aspects of managing *hynet* grid databases.

2.3.1 Creating a new Database

In this example, some changes are made to a scenario that are considered as modifications of the grid infrastructure and, subsequently, it is saved as a new grid. To this end, let us consider the following scenario.

```
scenario = ht.load_scenario(database, scenario_id=100)
print(ht.calc_opf(scenario).details)
```

In the injector result, we can observe that two generators operate at a very low power factor. To ensure that the power factor of the injectors is at least 0.9, we can modify their capability regions accordingly (see also *Visualization and Specification of Capability Regions*):

```
for cap in scenario.injector['cap']:
    cap.add_power_factor_limit(0.9)
print(ht.calc_opf(scenario).details)
```

If we now try to save this scenario (which we'll actually learn *below*), *hynet* issues an error as this is considered as a change of the grid infrastructure (For the sake of simplicity, scenarios may change capability regions only in size, but not in shape):

```
scenario.name = 'Nice power factor'
ht.save_scenario(database, scenario)
```

Thus, in order to store this scenario, we have to create a new database with this grid infrastructure. Assume the database shall be stored in `my_new_database.db` (which must not exist). We thus connect to this database and initialize it using

```
database_new = ht.connect('my_new_database.db')
scenario.grid_name = 'Hybrid PJM System with a Power Factor Limit'
ht.initialize_database(database_new, scenario)
```

Therewith, the grid infrastructure is stored to the database and, additionally, a scenario is created that captures the scenario information. Note that during the database initialization, the scenario ID and database URI in `scenario` are updated accordingly. If you are curious, you may browse the database, e.g., using the DB Browser for SQLite or SQLite Online.

2.3.2 Saving a Customized Scenario

This example assumes that *Creating a new Database* was finished before to have an example database at hand. Let us first connect to this database and check the scenarios therein:

```
database = ht.connect('my_new_database.db')
print(ht.get_scenario_info(database))
```

We see that there is one scenario available, which was created when we initialized the database, and load it using

```
scenario = ht.load_scenario(database)
```

Now, let's assume we want to create a scenario where injector 3 is decommitted (offline). To this end, we simply remove this injector from the injector data frame, update the scenario's name, and initiate the saving of the scenario, i.e.,

```
scenario.injector.drop(3, inplace=True)
scenario.name = "Injector 3 offline"
ht.save_scenario(database, scenario)
```

The saving procedure assigns a new ID to the scenario, saves the scenario information to the database, and updates the scenario ID and database URI in `scenario` accordingly. For the purpose of illustration, let's add another scenario that considers the outage of converter 2 at 80% of the original load:

```
scenario = ht.load_scenario(database)
scenario.converter.drop(2, inplace=True)
scenario.bus['load'] *= 0.8
scenario.name = "Outage of converter 2"
ht.save_scenario(database, scenario)
```

We can now see these scenarios when we inspect the database,

```
print(ht.get_scenario_info(database))
```

2.3.3 Removing Scenarios from a Database

This example assumes that *Saving a Customized Scenario* was finished before to have an example database at hand. In this database, we now find three scenarios:

```
database = ht.connect('my_new_database.db')
print(ht.get_scenario_info(database))
```

Let's assume we are not satisfied with the last two scenarios that we added and we want to remove them from the database. We can achieve this using

```
ht.remove_scenarios(database, scenario_ids=[1, 2])
```

2.3.4 Import Grid Data from the MATPOWER Format

hynet supports the import of MATPOWER test cases into the *hynet* grid database format, if the test case is stored as a MATPOWER test case struct `mpc` in a MATLAB MAT-file. For example, assume we want to import the MATPOWER test case `case5.m`. With MATPOWER properly installed, start MATLAB, load the test case, and save it to a MAT-file using

```
mpc = loadcase('case5.m');
save('case5.mat', 'mpc');
```

Then, open a terminal, navigate to the directory that contains this MAT-file, and perform the import using

```
python -m hynet import case5.mat
```

The corresponding *hynet* grid database is then saved as `case5.db`. To see more options for the import, type

```
python -m hynet import -h
```

For example, we can specify the output file name, the grid's name, and a description of the data using

```
python -m hynet import case5.mat -o pjm_system.db -g "PJM System" -d "This data␣
↪was imported from MATPOWER."
```

Remark: To perform the import from the Python shell, see `ht.import_matpower_test_case`.

Caution: It is important to be aware of the following particularities of the import:

1) The branch flow limit is interpreted differently in *hynet*, i.e., it is considered as an *ampacity rating* (thermal flow limit) stated as an MVA rating at 1 p.u. compared to the apparent power flow limit in MATPOWER. If the apparent power flow limit shall be enforced, the conservative substitute bounds described in Remark 1 in this paper can be utilized, which are applied to a scenario object `scenario` with `scenario.set_conservative_rating()`.

2) *hynet* exclusively employs *piecewise linear* (PWL) cost functions. Nonlinear polynomial cost functions in the MATPOWER test case are converted to PWL functions by sampling the polynomial equidistantly within the generator's active power bounds. The number of sample points can be specified with the option `-n`.

2.4 Construction of Hybrid AC/DC Grid Models

The following examples illustrate how HVDC systems may be added to a grid model. Please note that the presented modeling approach is targeted at system-level studies and does not explicitly model the transformer, filter, and phase reactor of VSC stations. The converters are parameterized with conversion loss factors of 1% and a Q/P capability ratio of 25%, which is rather conservative, cf. the brochure "HVDC Light - It's time to connect" of ABB AB (Revision H, 2017).

2.4.1 Converting AC Lines to DC Operation

For capacity expansion, the conversion of existing AC lines to DC operation can be an attractive option, as the construction of new transmission corridors is often difficult and protracted. The conversion can offer a significant increase of transmission capacity within the existing corridor, while the (VSC) converters additionally provide flexible power flow control and reactive power compensation which can additionally enhance the system's effective transmission capacity. In the following, it is illustrated how such a conversion may applied to a model by creating a MT-HVDC variant of the hybrid system presented in this paper. To this end, we first load the adapted PJM system:

```
database_acg = ht.connect('pjm_adapted.db')
acg = ht.load_scenario(database_acg)
print(acg)
```

The branches 5 and 6 connect the buses 3-4 and 4-5 and are converted to P2P-HVDC systems in the aforementioned work. Here, these two AC lines shall be converted to a 3-terminal HVDC system. We start by creating a copy of the original system and updating the meta data:

```
htg = acg.copy()
htg.grid_name = "Hybrid PJM System"
htg.description = htg.database_uri = ""
```

For the conversion to DC operation, *hynet* offers the following utilities:

```
help(ht.convert_ac_line_to_hvdc_system)
help(ht.convert_transformer_to_b2b_converter)
```

With the following code, the two AC lines are converted to an HVDC system. Please note that, like in the aforementioned work, the transmission capacity of the lines is intentionally *not* uprated to illustrate the increase of the effective transmission capacity due to the *flexibilization* of the grid.

```
ht.convert_ac_line_to_hvdc_system(
    htg,
    branch_id=5,
    loss_fwd=1.0,
    loss_bwd=1.0,
    q_to_p_ratio=0.25,
    base_kv_map={230.0: 325.0},
    capacity_factor=1.0
)

ht.convert_ac_line_to_hvdc_system(
    htg,
    branch_id=6,
    loss_fwd=1.0,
    loss_bwd=1.0,
    q_to_p_ratio=0.25,
    base_kv_map={230.0: 325.0},
    capacity_factor=1.0
)
```

The conversion introduces 3 DC buses and 3 AC/DC converters and updates the branch data, which can be seen by inspecting the scenario data:

```
print(htg)
```

Finally, let's analyze the impact of the conversion on the system's performance in terms of economic efficiency and transmission capacity. The following results illustrate that the flexibility introduced by the conversion of AC lines to DC operation can significantly improve the performance of a congested system, which confirms the findings in this paper. (In the latter, the results deviate due to different branch flow limits, see "*Import Grid Data from the MATPOWER Format*".)

2.4.1.1 Economic efficiency: Can this flexibilization measure reduce the injection costs?

```
cost = []
for scenario in [acg, htg]:
    result = ht.calc_opf(scenario)
    cost.append(result.get_total_injection_cost())
    print(result.details)
    print(result)
print("Injection cost reduction: {:.1%}".format(1 - cost[1] / cost[0]))
```

We can observe a significant reduction of the injection cost, which is due to the different utilization of injector 3 that exhibits the lowest marginal costs. In the original system, it's utilization is restricted by congestion, while it is operated at maximum capacity in the hybrid system. Note that the total losses in the hybrid system are higher than in the original system, which is not only due to the converter losses but also due to the "power flow routing" enabled by the grid flexibilization, where power may flow along (electrically) longer distances, cf. the branch result and losses for both systems. Here, the cost of the additional losses is outweighed by the improved utilization of the injectors.

2.4.1.2 Transmission capacity: Can this flexibilization measure increase the effective capacity?

```
loadability = []
for scenario in [acg, htg]:
    result = ht.calc_loadability(scenario)
    loadability.append(1 + result.load_increment_scaling)
    print(result)
print("Loadability improvement: {:.1%}".format(loadability[1] / loadability[0] -␣
↪1))
```

We can observe a significant improvement in the maximum loadability and, even more importantly, that the hybrid system is *not* limited by congestion but by the available generation capacity. This is even the case if the generation capacity is increased, e.g., by 20% via

```
for scenario in [acg, htg]:
    for cap in scenario.injector['cap']:
        cap.scale(1.2)
```

If the above experiment is repeated, it can be observed that the loadability improvement is substantially higher and that the hybrid system is still *not* limited by congestion.

2.4.2 Adding an HVDC System

In the following, it is illustrated how an HVDC system may be added to a grid model by exchanging the 3-terminal HVDC system by two P2P-HVDC systems. To this end, we first load the scenario from the database

```
scenario = ht.load_scenario(database)
```

where we can see that the 3-terminal HVDC system comprises the buses 6 to 8

```
scenario.get_dc_subgrids()
```

which are connected by the branches 5 and 6:

```
scenario.branch.loc[scenario.branch[['src', 'dst']].isin([6, 7, 8]).any(axis=
↪'columns')]
```

To set the scene, we backup the series resistance and rating of branch 6 and remove bus 8, which also removes all the connected entities:

```
z_bar, rating = scenario.branch.loc[6, ['z_bar', 'rating']]
scenario.remove_buses([8])
print(scenario)
```

We can see that a P2P-HVDC system between the AC buses 3-4 remains and we now introduce a P2P-HVDC system between the AC buses 4-5 by adding 2 DC buses, the respective AC/DC converters, and the DC line:

```
dc_bus_src = scenario.add_bus(
    type_=ht.BusType.DC,
    base_kv=325.0,
    v_min=0.9,
    v_max=1.1,
    zone=1
)

dc_bus_dst = scenario.add_bus(
    type_=ht.BusType.DC,
    base_kv=325.0,
    v_min=0.9,
    v_max=1.1,
    zone=1
)

cap_src = ht.ConverterCapRegion(
    p_bnd=[-rating, rating],
    q_bnd=[-0.25*rating, 0.25*rating]
)

cap_dst = ht.ConverterCapRegion(
    p_bnd=[-rating, rating]
```

(continues on next page)

(continued from previous page)

```
)

scenario.add_converter(
    src=4,
    dst=dc_bus_src,
    cap_src=cap_src.copy(),
    cap_dst=cap_dst.copy(),
    loss_fwd=1.0,
    loss_bwd=1.0
)

scenario.add_converter(
    src=5,
    dst=dc_bus_dst,
    cap_src=cap_src.copy(),
    cap_dst=cap_dst.copy(),
    loss_fwd=1.0,
    loss_bwd=1.0
)

scenario.add_branch(
    type_=ht.BranchType.LINE,
    src=dc_bus_src,
    dst=dc_bus_dst,
    z_bar=z_bar,
    rating=rating
)

scenario.grid_name += " (P2P-HVDC Systems)"
scenario.description = scenario.database_uri = ""
```

Concluding, let's verify and inspect the resulting scenario and compute an OPF:

```
scenario.verify()
print(scenario)
print(ht.calc_opf(scenario))
```

2.5 Feature- and Structure-Preserving Network Reduction

For large-scale power systems, repeated optimal power flow studies, e.g., in grid expansion planning, are often computationally highly demanding due to the extensive model complexity. To this end, *hynet* includes the feature- and structure-preserving network reduction proposed in this publication (preprint) to appropriately reduce the model complexity and streamline such studies. Based on a (peak load) reference scenario, this method utilizes a collection of topological, electrical, and market-based approaches to identify subgrids that are suitable for reduction, which are then selectively reduced while preserving the surrounding structure and designated features of the grid. In the following, the utilization of this network reduction method is illustrated by reproducing the results in this publication for the German grid (Please note that those results are based on the *hynet* Grid Database Library v1.0). It is assumed that **IPOPT is installed**, that a Python shell was started in a directory that contains the grid databases, and that the following commands were executed:

```
import hynet as ht
import hynet.reduction.large_scale as nr
database = ht.connect('germany2030nep.db')
```

Additionally, to track the progress, it is recommended to enable the logging of info messages via

```
import logging
logging.basicConfig(level=logging.INFO)
```

Remarks:

- In the parameter sweeps below, a *hynet* optimization server may be utilized. See also *Distributed Computation of Optimal Power Flows*.
- *hynet* also includes support for the reduction to a "copper plate" model, see `hynet.reduction.copper_plate`.

2.5.1 Selection of Reduction Parameters

This reduction method comprises a feature definition and three subsequent stages, i.e., topological, electrical, and market-based reduction, which are discussed below. Prior to these stages, the reference scenario is loaded and an OPF reference is computed:

```
scenario = ht.load_scenario(database)
opf_reference = ht.calc_opf(scenario.copy())
print(opf_reference)
```

2.5.1.1 1. Feature Definition

Features are entities in the model that are essential to the application-relevant accuracy and validity of the derived results and conclusions. To this end, the `bus` and `branch` data frame of the scenario is equipped with an additional Boolean-valued column `feature` that declares such features. In order to add the proposed features in this publication to the scenario, call

```
nr.add_standard_features(scenario, opf_reference)
```

The `feature` columns may also be modified to further customize the feature definition.

2.5.1.2 2. Topology-Based Reduction

In order to perform the topology based reduction, call

```
nr.reduce_by_topology(scenario)
```

This reduces single buses, lines of buses, and small "islands" at the boundary of the grid. To parameterize the "island" reduction process, please refer to the optional parameter `max_island_size`. The identification of "islands" can be computationally demanding and may be skipped using `max_island_size=0`. Finally, the reduction accuracy is evaluated by

```
evaluation = nr.evaluate_reduction(opf_reference, ht.calc_opf(scenario))
print(evaluation)
```

Please refer to the function's documentation for more information on the evaluation data.

2.5.1.3 3. Electrical Coupling-Based Reduction

This reduction step combines buses that are connected via a low series impedance, where "low" is defined by a threshold relative to the maximum series impedance modulus. To identify a proper parameterization, a sweep may be performed to select an adequate threshold:

```
evaluation = nr.sweep_rel_impedance_thres(scenario,
                                          opf_reference=opf_reference)
print(evaluation)
```

Using the generated visualization or tabular representation, a threshold may be selected. In the vicinity of injectors, this reduction may introduce more pronounced errors, due to which a heuristic feature refinement based on the depth to critical injectors should be considered. To this end, a more ambitious selection of the threshold may be made, for example `0.05`, and additional features may be added based on a certain depth from critical injectors. To identify a proper depth, an additional sweep is performed:

```
evaluation = nr.sweep_feature_depth(scenario,
                                    rel_impedance_thres=0.05,
                                    opf_reference=opf_reference)
print(evaluation)
```

Using the generated visualization or tabular representation, an appropriate depth may be selected. For example, `4` is appropriate here and the respective reduced scenario is extracted from the evaluation data using

```
scenario = evaluation.at[4, 'opf'].scenario
```

2.5.1.4 4. Market-Based Reduction

This approach reduces subgrids that exhibit a similar locational marginal price (LMP) or, more precisely, dual variable of the nodal active power balance. As the characteristic range for the LMP is rather specific to a scenario, it may be reasonable to inspect it beforehand (Please note that the LMP profile of this example is quite "messy" as some renewables with zero marginal cost are not fully utilized):

```
ht.show_lmp_profile(ht.calc_opf(scenario))
```

To select an appropriate price threshold, a parameter sweep may be performed, where the sweep range is based on the previously inspected LMP profile:

```
import numpy as np
sweep_values = list(np.around(np.linspace(0.02, 0.2, 10), decimals=2))
evaluation = nr.sweep_max_price_diff(scenario,
                                     values=sweep_values,
                                     opf_reference=opf_reference)
print(evaluation)
```

Using the generated visualization or tabular representation, an appropriate price threshold is selected, for example `0.08`. In the following, this parameter selection is utilized in a combined reduction process and the reduced model is stored to a grid database.

2.5.2 Performing and Saving a Network Reduction

Once the reduction parameters are selected, a combined reduction process may be initiated to generate the reduced model and to evaluate the individual reduction stages:

```
scenario = ht.load_scenario(database)
evaluation, bus_id_map = nr.reduce_system(scenario,
                                          rel_impedance_thres=0.05,
                                          feature_depth=4,
                                          max_price_diff=0.08,
                                          show_evaluation=True,
                                          return_bus_id_map=True,
                                          preserve_aggregation=True)
print(evaluation)
```

This function automatically adds the standard features prior to the reduction if no features are specified. If custom features are utilized, they must be defined prior to the call of `reduce_system`. After the reduction, `scenario` represents the reduced model and contains the information on aggregated buses in the column `aggregation` of the bus data frame, i.e., it specifies which buses were aggregated to the respective retained bus:

```
print(scenario.bus['aggregation'].head(n=10))
```

This aggregation information is additionally stored in the bus annotation to preserve it when the model is stored to a grid database.

```
print(scenario.bus['annotation'].head(n=10))
```

Note that the reduction process only removes buses and branches, while all other entities are retained with the same ID and may be updated, e.g., the terminal bus of injectors within a reduced subgrid is set to the respective representative bus.

Finally, the model's name is updated and the reduced model is stored to a new database, alongside the scenarios of the original system:

```
scenario.grid_name = 'Reduced ' + scenario.grid_name
database_nr = ht.connect('germany2030nep_nr.db')
ht.initialize_database(database_nr, scenario)
ht.copy_scenarios(database, database_nr, bus_id_map=bus_id_map)
```

2.5.3 Loading a Scenario of a Reduced Model

Loading a scenario of a reduced model follows the same procedure as with any other grid database, for example

```
database_nr = ht.connect('germany2030nep_nr.db')
scenario = ht.load_scenario(database_nr)
```

However, if the information on aggregated buses in the column `aggregation` of the bus data frame is required, it must be restored from the bus annotation:

```
nr.restore_aggregation_info(scenario)
```

2.6 Miscellaneous Aspects

2.6.1 Visualization and Specification of Capability Regions

As the specification and conception of injector and converter capability regions is somewhat intricate, *hynet* includes a GUI for their visualization. For this GUI, TkInter must be installed, which may be done e.g. with Conda by running

```
conda install tk
```

in the terminal. In case you are using MAC OS X, please be aware of this issue (Set `matplotlib`'s backend to `TkAgg` *before* importing *hynet*).

To demonstrate this GUI, open a Python shell, connect to the example database, and load the default scenario:

```
import hynet as ht
database = ht.connect('pjm_hybrid.db')
scenario = ht.load_scenario(database)
```

For example, to display the parameter description and edit the capability region of injector 1, call

```
help(ht.CapRegion)
scenario.injector.at[1, 'cap'].edit()
```

The GUI can also be used to display the operating point obtained by an OPF in the capability region. For example, for injector 2 this is achieved with

```
result = ht.calc_opf(scenario)
scenario.injector.at[2, 'cap'].show(result.injector.at[2, 's'])
```

CHAPTER 3

Package Documentation

3.1 Subpackages

3.1.1 hynet.data package

3.1.1.1 Submodules

3.1.1.2 hynet.data.connection module

Manage *hynet*'s database connections.

class hynet.data.connection.**DBConnection**(*database_uri*)
Bases: object

Manager for a *hynet* grid database connection.

See also:

hynet.data.connection.DBTransaction

description
Return the description of this database.

Before the text is returned, the description retrieved from the database is wrapped to an appropriate column width to improve readability.

empty
Return True if the database does not contain grid information.

get_setting(*key*)
Return the database setting for the specified key.

Parameters **key** (DBInfoKey) – The key for which the value shall be retrieved.

Returns **value** – The value associated with the provided key.

Return type str

Raises **ValueError** – If the setting was not found.

grid_name
Return the name of the grid in this database.

set_setting(*key*, *value*)
Set the database setting for the specified key.

Parameters

- **key** (DBInfoKey) – The key for which the value shall be set.
- **value** (*str*) – Value to be set.

Raises **ValueError** – If the value is not a string.

start_session()
Return a new database session as an SQLAlchemy Session object.

Remark: This function is for internal use.

version
Return the *hynet* grid database format version of this database.

class hynet.data.connection.**DBTransaction**(*database*)
Bases: object

Database transaction that is automatically committed at the exit block.

add(*object_*)
Add an object to the session.

add_all(*collection*)
Add a collection of objects to the session.

delete(*object_*)
Mark the object as deleted in the session.

delete_all(*collection*)
Mark a collection of objects as deleted

execute(*query*)
Execute the SQL expression construct or string statement.

query(*object_type*)
Return a new SQLAlchemy Query object for this session.

update(*object_*)
Update the state of the object in the session.

hynet.data.connection.**connect**(*database_uri*)
Return a connection to the specified *hynet* grid database.

Parameters **database_uri** (*str*) – URI or file name of the *hynet* grid database.

Returns **database** – Connection to the *hynet* grid database.

Return type *DBConnection*

3.1.1.3 hynet.data.example_days module

Support for the generation of scenarios of exemplary days.

hynet.data.example_days.**add_example_days**(*database*, *base_case*)
Add the exemplary day scenarios to the database.

hynet.data.example_days.**initialize_database_with_example_days**(*database*, *base_case*)
Initialize the database with the base case and add exemplary day scenarios.

The added day scenarios are exemplary winter and summer weekdays/weekends obtained by scaling the base case load according to the data provided in [1], Table 4.

Parameters

- **database** (DBConnection) – Connection to the destination database. This database must be empty.
- **base_case** (Scenario) – Base case scenario for the new database.

Raises **ValueError** – In case that the destination database is not empty or the provided scenario data exhibits integrity or validity issues.

References

3.1.1.4 hynet.data.import_ module

Import of model data into the *hynet* data format.

class hynet.data.import_.**BranchConstants**

Bases: object

Constants for the MATPOWER branch data, see the MATPOWER manual.

ANGMAX = 12

ANGMIN = 11

BR_B = 4

BR_R = 2

BR_STATUS = 10

BR_X = 3

F_BUS = 0

MU_ANGMAX = 20

MU_ANGMIN = 19

MU_SF = 17

MU_ST = 18

PF = 13

PT = 15

QF = 14

QT = 16

RATE_A = 5

RATE_B = 6

RATE_C = 7

SHIFT = 9

TAP = 8

T_BUS = 1

class hynet.data.import_.**BusConstants**

Bases: object

Constants for the MATPOWER bus data, see the MATPOWER manual.

BASE_KV = 9

BS = 5

BUS_AREA = 6

BUS_I = 0

BUS_TYPE = 1

GS = 4

LAM_P = 13

LAM_Q = 14

MU_VMAX = 15

MU_VMIN = 16

NONE = 4

PD = 2

PQ = 1

PV = 2

QD = 3

REF = 3

VA = 8

VM = 7

VMAX = 11

VMIN = 12

ZONE = 10

class hynet.data.import_.**DCLineConstants**

Bases: object

Constants for the MATPOWER DC line data, see the MATPOWER manual.

BR_STATUS = 2

F_BUS = 0

LOSS0 = 15

LOSS1 = 16

MU_PMAX = 18

MU_PMIN = 17

MU_QMAXF = 20

MU_QMAXT = 22

MU_QMINF = 19

MU_QMINT = 21

PF = 3

PMAX = 10

PMIN = 9

PT = 4

QF = 5

QMAXF = 12

QMAXT = 14

QMINF = 11

QMINT = 13

QT = 6

T_BUS = 1

VF = 7

VT = 8

class hynet.data.import_.**GeneratorConstants**

Bases: object

Constants for the MATPOWER generator data, see the MATPOWER manual.

APF = 20

COST = 4

GEN_BUS = 0

GEN_STATUS = 7

MBASE = 6

MODEL = 0

MU_PMAX = 21

MU_PMIN = 22

MU_QMAX = 23

MU_QMIN = 24

NCOST = 3

PC1 = 10

PC2 = 11

PG = 1

PMAX = 8

PMIN = 9

POLYNOMIAL = 2

PW_LINEAR = 1

QC1MAX = 13

QC1MIN = 12

QC2MAX = 15

QC2MIN = 14

QG = 2

QMAX = 3

QMIN = 4

RAMP_10 = 17

RAMP_30 = 18

RAMP_AGC = 16

RAMP_Q = 19

SHUTDOWN = 2

STARTUP = 1

VG = 5

hynet.data.import_.**import_matpower_test_case**(*input_file*, *output_file=None*, *grid_name=None*, *description=''*, *num_sample_points=10*, *res_detection=False*)

Import a MATPOWER test case file into *hynet*'s database format.

This function imports a MATPOWER test case, which is stored as a MATPOWER test case struct `mpc` in a MATLAB MAT-file, to a *hynet*'s database. To prepare the import, start MATLAB and perform the following two steps:

1) Load the MATPOWER test case into the variable `mpc`:

```
mpc = loadcase('your_matpower_test_case_file.m');
```

2) Save the MATPOWER test case struct `mpc` to a MATLAB MAT file:

```
save('your_matpower_test_case_file.mat', 'mpc');
```

Call this function with the MAT-file as the input.

Parameters

- **input_file** (*str*) – MATLAB MAT-file (.mat) with the MATPOWER test case struct `mpc`.
- **output_file** (*str, optional*) – Destination *hynet* grid database file. By default (None), the file name is set to the input file name with a `.db` extension.
- **grid_name** (*str, optional*) – Name of the grid. By default (None), the grid name is set to the input file name excluding the extension.
- **description** (*str, optional*) – Description of the grid model and, if applicable, copyright and licensing information.
- **num_sample_points** (*int, optional*) – Number of sample points that shall be used for the conversion of polynomial to piecewise linear cost functions (on the interval from minimum to maximum output). This setting is a trade-off between an accurate representation of the original cost function and the number of additional constraints in the OPF problem.
- **res_detection** (*bool, optional*) – Detection of the injector type for renewable energy sources (RES), which is inactive by default. This scheme is motivated by the German grid data, which contains PV- and wind-based injectors with zero marginal cost that are arranged in a specific pattern: The injectors that connect to the same bus are stored consecutively in the generator matrix. If two or more injectors connect to the same bus, then the *last* one is wind-based and the *second to the last* is PV-based generation if their marginal cost is zero. If this parameter is set to True, this RES detection scheme is enabled.

Returns **output_file** – Destination *hynet* grid database file name.

Return type str

Raises

- **ValueError** –
- **NotImplementedError** –

3.1.1.5 hynet.data.interface module

Interface to access the model data in a *hynet* grid database.

hynet.data.interface.**copy_scenarios**(*source_database*, *destination_database*, *scenario_ids=None*, *bus_id_map=None*)

Copy the scenarios from the source to the destination *hynet* grid database.

Caution: Note that this function does *not* verify that the grid infrastructure of the source and destination database is compatible, it only copies the content of the scenario information tables for the requested scenarios. In case a scenario with a specified ID already exists in the destination database, it is *skipped*.

Parameters

- **source_database** (DBConnection) – Connection to the source *hynet* grid database.
- **destination_database** (DBConnection) – Connection to the destination *hynet* grid database.
- **scenario_ids** (*list[hynet_id_], optional*) – List of scenario identifiers that shall be copied. By default, all scenarios are copied.
- **bus_id_map** (*pandas.Series, optional*) – If provided, this series, *indexed by the bus IDs in the source database*, specifies the mapping of a bus ID in the source database to a bus ID in the destination database. This is necessary, e.g., after a network reduction, where certain buses are combined and the load shall be accumulated at the aggregated bus. By default, a one-to-one mapping is considered.

hynet.data.interface.**fix_hynet_id**(*series*)

Fix the data type of the given nullable integer pandas series.

When reading integer columns that contain NULL values using pandas.read_sql_table, they are converted to float [1]. This may lead to errors, e.g. the PWL function foreign keys cause key errors during indexing. This function fixes the type.

References

hynet.data.interface.**get_max_bus_id**(*database*)

Determine the highest bus ID in the specified *hynet* grid database.

Parameters **database** (DBConnection) – Connection to the *hynet* grid database.

Returns **result** – The highest bus ID in the database or None if there are no buses.

Return type hynet_id_ or None

hynet.data.interface.**get_max_scenario_id**(*database*)

Determine the highest scenario ID in the specified *hynet* grid database.

Parameters **database** (DBConnection) – Connection to the *hynet* grid database.

Returns **result** – The highest scenario ID in the database or None if there are no scenarios.

Return type hynet_id_ or None

hynet.data.interface.**get_scenario_info**(*database*)

Load a list of all scenarios from the specified *hynet* grid database.

Parameters **database** (DBConnection) – Connection to the *hynet* grid database.

Returns

Data frame with a list of all scenarios, indexed by the *scenario ID*, which comprises the following columns:

name: (str) Name of the scenario.

time: (hynet_float_) Relative time in hours of the scenario (w.r.t. the scenario group start time).

annotation: (str) Annotation string for supplementary information.

Return type pandas.DataFrame

Raises **ValueError** – In case that the database is empty.

hynet.data.interface.**initialize_database**(*database*, *scenario*)
Initialize an empty *hynet* grid database with the data of the scenario.

The database is populated with the grid infrastructure and scenario data of the given scenario object. In the given scenario object, the scenario ID is reset to `0` and the database URI is updated.

Parameters

- **database** (`DBConnection`) – Connection to the *hynet* grid database.
- **scenario** (`hynet.scenario.representation.Scenario`) – Scenario object with the grid infrastructure and scenario data.

Raises **ValueError** – In case the database is not empty or any kind of data integrity or validity violation is detected.

hynet.data.interface.**load_scenario**(*database*, *scenario_id=0*)
Load the specified scenario from the *hynet* grid database.

Parameters

- **database** (`DBConnection`) – Connection to the *hynet* grid database.
- **scenario_id** (*hynet_id_, optional*) – Identifier of the scenario; default is `0`.

Returns **scenario** – Scenario object with the loaded data.

Return type *hynet.scenario.representation.Scenario*

Raises **ValueError** – In case that the database is empty or the scenario with the specified ID was not found.

hynet.data.interface.**remove_scenarios**(*database*, *scenario_ids*)
Remove the specified scenarios from the *hynet* grid database file.

Parameters

- **database** (`DBConnection`) – Connection to the *hynet* grid database.
- **scenario_ids** (*list[hynet_id_]*) – List of identifiers for the scenarios that shall be removed.

Raises **ValueError** – In case that a scenario with the specified ID was not found.

hynet.data.interface.**save_scenario**(*database*, *scenario*, *auto_id=True*)
Save the provided scenario to the specified *hynet* grid database.

The scenario information in the provided scenario object is extracted and saved to the specified *hynet* grid database. Note that a scenario can only capture the following changes:

- Changes of the load (`scenario.bus['load']`)
- Selected changes of injectors:
  - Scaling of the cost function for active and reactive power (`scenario.injector[['cost_p', 'cost_q']]`). The scaling must be the same for both cost functions of an injector.
  - Changes of the box constraints of the capability region (`p_min`, `p_max`, `q_min`, and `q_max` of the objects in `scenario.injector['cap']`). Note that changes of the parameters of the half-spaces are *not* supported.)
- Inactivity of buses, branches, converters, injectors, and shunts. For example, to deactivate (decommit) an injector, remove (drop) the respective row in the `injector` data frame of the scenario.

Any other changes are considered as a modification of the grid infrastructure and cannot be captured as a scenario.

Parameters

- **database** (`DBConnection`) – Connection to the *hynet* grid database.

- **scenario** (Scenario) – Scenario object that represents the new scenario.
- **auto_id** (*bool, optional*) – If True (default), the scenario ID is set to the lowest available scenario ID in the database. Otherwise, the scenario ID of the Scenario object is applied.

Raises **ValueError** – In case the database is empty or any kind of data integrity or validity violation is detected.

See also:

hynet.data.interface.initialize_database()

3.1.1.6 hynet.data.structure module

Schema and SQLAlchemy declaratives for *hynet* grid databases.

class hynet.data.structure.**DBBranch**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a branch.

angle_max

angle_min

annotation

b_dst

b_src

drop_max

drop_min

dst

dst_id

id

length

phase_dst

phase_src

r

rating

ratio_dst

ratio_src

src

src_id

type

x

class hynet.data.structure.**DBBus**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a bus.

annotation

base_kv

id

ref

type

v_max

v_min

zone

class hynet.data.structure.**DBCapabilityRegion**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a capability region.

annotation

id

lb_ofs

lb_slp

lt_ofs

lt_slp

p_max

p_min

q_max

q_min

rb_ofs

rb_slp

rt_ofs

rt_slp

class hynet.data.structure.**DBConverter**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a converter.

annotation

cap_dst

cap_dst_id

cap_src

cap_src_id

dst

dst_id

id

loss_bwd

loss_fix

loss_fwd

src

src_id

class hynet.data.structure.**DBInfo**(**kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a *hynet* grid database setting.

key

value

class hynet.data.structure.**DBInjector**(**kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for an injector.

annotation

bus

bus_id

cap

cap_id

cost_p

cost_p_id

cost_q

cost_q_id

cost_start

cost_stop

energy_max

energy_min

id

min_down

min_up

ramp_down

ramp_up

type

class hynet.data.structure.**DBSamplePoint**(**kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a cost function sample point.

id

x

y

class hynet.data.structure.**DBScenario**(**kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a scenario.

annotation

id

loss_price

name

time

class hynet.data.structure.**DBScenarioInactivity**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for the inactivity specification in a scenario.

entity_id

entity_type

scenario

scenario_id

class hynet.data.structure.**DBScenarioInjector**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for the injector adaptation in a scenario.

cost_scaling

injector

injector_id

p_max

p_min

q_max

q_min

scenario

scenario_id

class hynet.data.structure.**DBScenarioLoad**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a load in a scenario.

bus

bus_id

p

q

scenario

scenario_id

class hynet.data.structure.**DBShunt**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Dataset for a shunt.

annotation

bus

bus_id

id

p

q

3.1.1.7 Module contents

3.1.2 hynet.distributed package

3.1.2.1 Submodules

3.1.2.2 hynet.distributed.client module

hynet optimization client for distributed computation.

`hynet.distributed.client.`**`start_optimization_client`**(*server_ip*, *port=None*, *authkey=None*, *num_workers=None*, *verbose=True*)

Create, connect, and start a *hynet* optimization client.

Note that this call is blocking until the *hynet* optimization server, to which the client is connected, is shut down.

Parameters

- **`server_ip`** (*`str`*) – IP address of the *hynet* optimization server.
- **`port`** (*`int, optional`*) – TCP port of the *hynet* optimization server.
- **`authkey`** (*`str, optional`*) – Authentication key for the *hynet* optimization server.
- **`num_workers`** (*`int, optional`*) – Number of worker processes that should run in parallel. If more than one worker is started, it is recommended to disable the internal parallel processing, see `parallelize` in `hynet.config`.
- **`verbose`** (*`bool, optional`*) – If True (default), some information on activity of the client is printed to the standard output.

3.1.2.3 hynet.distributed.server module

hynet optimization server for distributed computation.

`class` `hynet.distributed.server.`**`OptimizationJob`**(*data*, *solver=None*, *solver_type=<SolverType.QCQP: 'QCQP'>*)

Bases: `object`

Represents a *hynet* optimization job.

Customization: To customize the job processing, derive from this class and override `process`.

Parameters

- **`data`** (`Scenario or SystemModel or QCQP`) – Scenario (to solve its OPF), a problem builder (an object of a derived class of `SystemModel` like `OPFModel`), or a QCQP specification.
- **`solver`** (`SolverInterface, optional`) – Solver for the provided problem. The default automatically selects an appropriate solver of the specified solver type. Please make sure that the selected solver is installed on all client machines.
- **`solver_type`** (`SolverType, optional`) – Solver type for the automatic solver selection (default `SolverType.QCQP`). It is ignored if `solver` is not `None`.

`process`()

Process the optimization job and return the result (or exception).

`class` `hynet.distributed.server.`**`OptimizationServer`**(*port*, *authkey*, *local*)

Bases: `object`

hynet optimization server for distributed computation.

This server manages the distributed computation of a set of *hynet* optimization problems (OPF or QCQPs) on *hynet* optimization clients.

calc_jobs(*job_list*, *solver=None*, *show_progress=True*)
Calculate the list of *hynet* optimization jobs and return the results.

The provided list of jobs is processed by distributing them to the connected *hynet* optimization clients, collecting the results, and returning an array of results that corresponds with the provided array of jobs. Note that if there are no clients connected, this method will wait until a client is connected to process the jobs.

Parameters

- **job_list** (*array-like*) – List of *hynet* optimization jobs (`OptimizationJob`) or problem specifications (`Scenario` [issues an OPF computation], `SystemModel`, or `QCQP`).
- **solver** (`SolverInterface`, *optional*) – If provided, this solver is used for problem specifications (`Scenario`, `SystemModel`, or `QCQP`). It is ignored for job specifications (`OptimizationJob`).
- **show_progress** (*bool, optional*) – If True (default), the progress is reported to the standard output.

Returns **results** – Array containing the optimization results.

Return type numpy.ndarray

shutdown()
Stop the *hynet* optimization server and all connected clients.

start_clients(*client_list*, *server_ip*, *ssh_user=None*, *ssh_port=None*, *num_workers=None*, *log_file=None*, *suppress_output=True*)
Automated start of *hynet* optimization clients.

This method provides an automatic start of *hynet* optimization clients via SSH if the server can connect to the clients via `ssh [client]` (e.g. by configuring SSH keys; please be aware of the related aspects of system security). *hynet* must be properly installed on all client machines.

This function uses SSH to run the *hynet* package with the sub-command `client` and corresponding command line arguments (`python -m hynet client ...`) on every client machine. To customize the SSH and Python command, see `hynet.config`.

Parameters

- **client_list** (*array-like*) – List of strings containing the host names or IP addresses of the client machines.
- **server_ip** (*str*) – IP address the *hynet* optimization server.
- **ssh_user** (*str, optional*) – The user name for the SSH login on the client machines. By default, this is set to the current user name (`getpass.getuser()`).
- **ssh_port** (*int, optional*) – Port on which SSH is running on the client machines.
- **num_workers** (*int, optional*) – Number of worker processes that should run in parallel on every client machine.
- **log_file** (*str, optional*) – Log file on the client machines to capture the output.
- **suppress_output** (*bool, optional*) – If `True` (default), the activity output of the optimization clients is suppressed.

hynet.distributed.server.**start_optimization_server**(*port=None*, *authkey=None*, *local=False*)

Create, start, and return a *hynet* optimization server.

Parameters

- **port** (*int, optional*) – TCP port on which the *hynet* optimization server shall be running.
- **authkey** (*str, optional*) – Authentication key that must be presented by *hynet* optimization clients to connect to the server.
- **local** (*bool, optional*) – If `True` (default is `False`), the optimization server processes all jobs on the local machine and *connections of clients are not accepted*. In case that some code is designed to utilize distributed computation, but the the server cluster is not available, this local mode supports the computation on the local machine without the client management overhead.

Returns **server** – The *hynet* optimization server.

Return type *OptimizationServer*

3.1.2.4 Module contents

Distributed computation functionality in *hynet*.

3.1.3 hynet.expansion package

3.1.3.1 Submodules

3.1.3.2 hynet.expansion.conversion module

Utilities for the conversion of AC lines and transformers to DC operation.

hynet.expansion.conversion.**convert_ac_line_to_hvdc_system**(*scenario*, *branch_id*, *loss_fwd*, *loss_bwd*, *q_to_p_ratio*, *base_kv_map=None*, *capacity_factor=nan*, *amalgamate=True*)

Convert the specified AC line to an HVDC system.

This function models the conversion of an AC line to DC operation by introducing AC/DC converters at the line's terminals and updating the branch parameters. If `amalgamate` is `True` (default) and there is already an (appropriate) HVDC system connected to either or both terminals of the line, the converted line is connected to this DC subgrid (to reduce the number of converters). Otherwise, always a point-to-point HVDC system is implemented. The branch parameters are updated by retaining only the series resistance and line rating, where both are updated according to the change of the base voltage. For the latter, the uprating may also be specified explicitly via `capacity_factor`. The introduced DC buses inherit the voltage limits (in p.u.) and zone from the respective AC bus. Please note that this is a simplified representation of such a conversion process for use in system-level studies, while the actual conversion of an AC line to DC operation involves extensive and case-specific considerations dependent on the present line configuration, see e.g. [1] for more details.

Parameters

- **scenario** (`Scenario`) – Scenario that shall be modified.
- **branch_id** (*hynet_id_*) – ID of the branch that shall be converted.

- **loss_fwd** (*float*) – Loss factor in percent for the *forward* flow of active power of the introduced converters.
- **loss_bwd** (*float*) – Loss factor in percent for the *backward* flow of active power of the introduced converters.
- **q_to_p_ratio** (*float*) – Ratio of the Q-capability of the introduced converters w.r.t. their P-capability.
- **base_kv_map** (*dict[hynet_float_, hynet_float_,], optional*) – Dictionary that maps the AC base voltage (in kV) to the DC base voltage (in kV). By default, a mapping `x -> floor(sqrt(2) * x)` is used.
- **capacity_factor** (*float, optional*) – Capacity factor for the conversion, i.e., the branch rating and the converter P-capability is set to the current branch rating times the capacity factor. By default, the capacity factor is set to the uprating due to the change of the base voltage.
- **amalgamate** (*bool, optional*) – If `True` (default), the converted line is amalgamated with an adjacent HVDC system if possible. Otherwise, always a point-to-point HVDC system is implemented. By convention, only AC/DC converters with the source terminal at the AC side are considered in the detection of and amalgamation with adjacent HVDC systems. The converters introduced by this function adhere to this convention.

Returns

- **converter_id_src** (*.hynet_id_*) – ID of the AC/DC converter at the source terminal of the converted line.
- **converter_id_dst** (*.hynet_id_*) – ID of the AC/DC converter at the destination terminal of the converted line.
- **amalgamated** (*tuple(bool, bool)*) – Named tuple `('src', 'dst')` with a Boolean flag for the source and destination terminal of the line that is `True` if that terminal was amalgamated with an existing HVDC system and `False` otherwise.

References

hynet.expansion.conversion.**convert_transformer_to_b2b_converter**(*scenario*, *branch_id*, *loss_dyn*, *q_to_p_ratio*, *capacity_factor=1.0*, *amalgamate=True*, *annotation_separator=', '*)

Replace the specified transformer (or AC branch) by an AC/AC converter.

This function models the conversion of a transformer to DC operation by replacing the branch with an AC/AC (back-to-back) converter. Please note that this is a simplified representation of such a conversion process for use in system-level studies.

Parameters

- **scenario** (Scenario) – Scenario that shall be modified.
- **branch_id** (*hynet_id_*) – ID of the branch that shall be converted.
- **loss_dyn** (*float*) – Loss factor in percent for the forward and backward conversion losses of the introduced converter.

- **q_to_p_ratio** (*float*) – Ratio of the Q-capability of the introduced converter w.r.t. its P-capability.
- **capacity_factor** (*float, optional*) – Capacity factor for the conversion (default `1.0`): The P-capability is set to the branch rating times the capacity factor.
- **amalgamate** (*bool, optional*) – If `True` (default) and there exists an AC/AC converter between the source and destination bus of the branch, this converter is uprated accordingly. Otherwise, always a new converter is added.
- **annotation_separator** (*str, optional*) – If not `None` (default `', '`), the branch annotation is copied to the converter annotation and, if the latter is not empty, this separator string is employed to separate the annotations.

Returns

- **converter_id** (*hynet_id_*) – ID of the AC/AC converter that replaces the branch.
- **amalgamated** (*bool*) – `True` if amalgamated and `False` otherwise.

3.1.3.3 hynet.expansion.selection module

Utilities for the selection of branches for an AC to DC conversion.

`hynet.expansion.selection.`**get_branches_outside_mst**(*scenario*, *branch_weights*)

Return an array of branch IDs that reside outside the minimum spanning tree.

The branch weights associate a weight with (selected) branches of the scenario and all *corridors* defined by these branches are considered as edges of the graph. Please note that converters are not considered, i.e., the function operates on (selected) branches of the subgrids. This function returns the IDs of the branches that reside *outside* the corridors of the minimum spanning tree.

Parameters

- **scenario** (Scenario) – Scenario that shall be considered.
- **branch_weights** (*pandas.Series*) – Branch weights indexed by the branch ID.

Returns **index** – Pandas index with the branch IDs of those branches that reside *outside* the corridors of the minimum spanning tree.

Return type pandas.Index

See also:

get_series_resistance_weights()

`hynet.expansion.selection.`**get_islanding_branches**(*scenario*, *show_progress=True*)

Return the branch IDs for all corridors whose removal leads to islanding.

Branches which are part of *all* spanning trees of (the corridors of) a grid are of particular interest, because their congestion can directly lead to infeasibility of the optimal power flow as there are no alternative power flow routes. This function identifies the corridors that are part of all spanning trees and returns the IDs of the branches that reside in these corridors.

Parameters

- **scenario** (Scenario) – Scenario that shall be evaluated.
- **show_progress** (*bool, optional*) – If `True` (default), the progress is reported to the standard output.

Returns Index with the ID of all branches that reside in corridors whose removal leads to islanding.

Return type pandas.Index

hynet.expansion.selection.**get_mst_branches**(*scenario*, *branch_weights*)

Return an array of branch IDs that are part of the minimum spanning tree.

The branch weights associate a weight with (selected) branches of the scenario and all *corridors* defined by these branches are considered as edges of the graph. Please note that converters are not considered, i.e., the function operates on (selected) branches of the subgrids. This function returns the IDs of the branches that reside in the corridors of the minimum spanning tree.

Parameters

- **scenario** (Scenario) – Scenario that shall be considered.
- **branch_weights** (*pandas.Series*) – Branch weights indexed by the branch ID.

Returns **index** – Pandas index with the branch IDs of those branches that reside in the corridors of the minimum spanning tree.

Return type pandas.Index

See also:

get_series_resistance_weights()

hynet.expansion.selection.**get_series_resistance_weights**(*scenario*, *prefer_transformers=False*)

Return a pandas Series of branch weights based on their series resistance in p.u.

Parameters

- **scenario** (Scenario) – Scenario for which the branch weights shall be created.
- **prefer_transformers** (*bool, optional*) – If `True` (default `False`), the weight of transformer branches is increased by the maximum series resistance in the system to prefer their conversion to DC operation.

Returns **branch_weights** – Branch weights indexed by the branch ID.

Return type pandas.Series

See also:

get_branches_outside_mst()

3.1.3.4 Module contents

Collection of utilities for grid expansion measures.

3.1.4 hynet.loadability package

3.1.4.1 Submodules

3.1.4.2 hynet.loadability.calc module

Calculation of the maximum loadability.

hynet.loadability.calc.**calc_loadability**(*data*, *scenario_id=0*, *solver=None*, *solver_type=<SolverType.QCQP: 'QCQP'>*, *initial_point_generator=None*, *converter_loss_error_tolerance=0.0005*)

Calculate the maximum loadability.

This function formulates and solves the maximum loadability problem as defined in equations (1) - (3) in [1] based on the feasibility set of the OPF problem in *hynet*, i.e., *hynet*'s the power balance equations are extended with a scaled load increment and the scaling of the increment is maximized. The nodal load increment is defined by the column `'load_increment'` in the `bus` data frame of the scenario. If this

column is not present, the load increment is set to the nodal load (i.e., the `'load'` column of the `bus` data frame), which maintains a constant power factor at the loads.

Parameters

- **data** (*DBConnection or Scenario or LoadabilityModel*) – Connection to a *hynet* grid database, a `Scenario` object, or a `LoadabilityModel` object.
- **scenario_id** (*hynet_id_, optional*) – Identifier of the scenario. This argument is ignored if `data` is a `Scenario` or `LoadabilityModel` object.
- **solver** (*SolverInterface, optional*) – Solver for the QCQP problem; the default automatically selects an appropriate solver of the specified solver type.
- **solver_type** (*SolverType, optional*) – Solver type for the automatic solver selection (default `SolverType.QCQP`). It is ignored if `solver` is not `None`.
- **initial_point_generator** (*InitialPointGenerator or None, optional*) – Initial point generator for QCQP solvers (ignored for relaxation-based solvers). By default (`None`), the initial point generation is skipped.
- **converter_loss_error_tolerance** (*hynet_float_, optional*) – Tolerance for the converter loss error in MW (default `5e-4`). If `None`, the loadability model's (default) setting is retained.

Returns **result** – Solution of the maximum loadability problem.

Return type *LoadabilityResult*

See also:

hynet.scenario.representation.Scenario(), *hynet.loadability.result.LoadabilityResult()*

References

3.1.4.3 hynet.loadability.model module

Model to evaluate the maximum loadability of a system with *hynet*.

class hynet.loadability.model.**LoadabilityModel**(*scenario*, *verify_scenario=True*)

Bases: *hynet.system.model.SystemModel*

Maximum loadability model for a steady-state scenario of a grid.

Based on the specification of a scenario via a `Scenario` object, this class provides the methods to generate a quadratically constrained quadratic program (QCQP) that captures the maximum loadability problem. The maximum loadability problem is considered as in equations (1) - (3) in [1] based on the feasibility set of the OPF problem in *hynet*, i.e., *hynet*'s the power balance equations are extended with a scaled load increment and the scaling of the increment is maximized. The nodal load increment is defined by the column `'load_increment'` in the `bus` data frame of the scenario. If this column is not present, the load increment is set to the nodal load (i.e., the `'load'` column of the `bus` data frame), which maintains a constant power factor at the loads.

See also:

hynet.scenario.representation.Scenario Specification of a steady-state grid scenario.

hynet.loadability.calc.calc_loadability Calculate the maximum loadability.

References

create_result(*qcqp_result*, *total_time=nan*, *qcqp_result_pre=None*)

Create and return a loadability result object.

Parameters

- **qcqp_result** (hynet.qcqp.result.QCQPResult) – Solution of the loadability QCQP.
- **total_time** (*hynet_float_, optional*) – Total time for solving the loadability problem.
- **qcqp_result_pre** (QCQPResult, *optional*) – Pre-solution of the loadability QCQP for converter mode detection.

Returns result

Return type *hynet.loadability.result.LoadabilityResult*

dim_z
Return the dimension of the state variable z.

The loadability problem only requires a single auxiliary variable, which is the nonnegative scaling factor for the load increment.

get_balance_constraints()
Return the active and reactive power balance constraints.

For the loadability problem, the power balance equations are augmented by a scaled load increment term, where the scaling is maximized by the objective.

get_z_bounds()
Return the auxiliary variable bounds z_lb <= z <= z_ub.

The lower is set to zero and the upper bound is omitted.

3.1.4.4 hynet.loadability.result module

Representation of a maximum loadability result.

class hynet.loadability.result.**LoadabilityResult**(*model*, *qcqp_result*, *total_time=nan*, *qcqp_result_pre=None*)

Bases: *hynet.system.result.SystemResult*

Result of a maximum loadability calculation.

Remark: In the data frames below, the respective column for the dual variables of a type of constraint (e.g., voltage drop) is only present if at least one constraint of this constraint type appears in the problem formulation.

Parameters

- **model** (LoadabilityModel) – Model for the processed maximum loadability problem.
- **empty** (*bool*) – True if the object does not contain any result data and False otherwise.
- **solver** (SolverInterface) – Solver object by which the result was obtained.
- **solver_status** (SolverStatus) – Status reported by the solver.
- **solver_time** (*float*) – Duration of the call to the solver in seconds.
- **optimal_value** (*float*) – Optimal objective value or numpy.nan if the solver failed.
- **total_time** (*float or numpy.nan*) – Total time for the loadability calculation, including the modeling, solving, and result assembly. If not provided, this time is set to numpy.nan.

- **reconstruction_mse** (*float*) – Unavailable if the result is empty and, otherwise, the mean squared error of the reconstructed bus voltages in case of a relaxation and `numpy.nan` otherwise.
- **load_increment_scaling** (*float*) – Maximum load increment scaling (cf. λ in equation (1) and (2) in [1]) or `numpy.nan` if the solver failed.
- **bus** (*pandas.DataFrame, optional*) – Unavailable if the result is empty and, otherwise, a data frame with the bus result data, indexed by the *bus ID*, which comprises the following columns:

  v: (**hynet_complex_**) Bus voltage rms phasor (AC) or bus voltage magnitude (DC).

  s_shunt: (**hynet_complex_**) Shunt apparent power in MVA. The real part constitutes the shunt losses in MW and the *negated* imaginary part constitutes the reactive power *injection*.

  bal_err: (**hynet_complex_**) Power balance residual in MVA, i.e., the evaluation of the complex-valued power balance equation at the system state. Theoretically, this should be identical to zero, but due to a limited solver accuracy and/or inexactness of the relaxation it is only approximately zero. This residual supports the assessment of solution accuracy and validity.

  dv_bal_p: (**hynet_float_**) Dual variable or KKT multiplier of the active power balance constraint.

  dv_bal_q: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power balance constraint.

  dv_v_min: (**hynet_float_**) Dual variable or KKT multiplier of the voltage lower bound.

  dv_v_max: (**hynet_float_**) Dual variable or KKT multiplier of the voltage upper bound.

- **branch** (*pandas.DataFrame, optional*) – Unavailable if the result is empty and, otherwise, a data frame with the branch result data, indexed by the *branch ID*, which comprises the following columns:

  s_src: (**hynet_complex_**) Apparent power flow in MVA at the source bus (measured as a flow *into* the branch).

  s_dst: (**hynet_complex_**) Apparent power flow in MVA at the destination bus (measured as a flow *into* the branch).

  i_src: (**hynet_complex_**) Current flow in p.u. at the source bus (measured as a flow *into* the branch).

  i_dst: (**hynet_complex_**) Current flow in p.u. at the destination bus (measured as a flow *into* the branch).

  v_drop: (**hynet_float_**) Relative voltage magnitude drop from the source bus to the destination bus.

  angle_diff: (**hynet_float_**) Bus voltage angle difference in degrees between the source and destination bus.

  effective_rating: (**hynet_float_**) Ampacity in terms of a long-term MVA rating at the *actual* bus voltage. If no rating is available, it is set to `numpy.nan`.

  rel_err: (**hynet_float_**) Branch-related relative reconstruction error $\kappa_k(V^\star)$ as defined in equation (24) in [2] in case of a relaxed QCQP or `numpy.nan` otherwise.

  dv_i_max_src: (**hynet_float_**) Dual variable or KKT multiplier of the ampacity constraint at the source bus or `numpy.nan` if unavailable.

dv_i_max_dst: (**hynet_float_**) Dual variable or KKT multiplier of the ampacity constraint at the destination bus or `numpy.nan` if unavailable.

dv_angle_min: (**hynet_float_**) Dual variable or KKT multiplier of the angle difference lower bound constraint or `numpy.nan` if unavailable.

dv_angle_max: (**hynet_float_**) Dual variable or KKT multiplier of the angle difference upper bound constraint or `numpy.nan` if unavailable.

dv_real_part: (**hynet_float_**) Dual variable or KKT multiplier of the +/-90 degrees constraint on the angle difference (cf. equation (27) in [3]) or `numpy.nan` if unavailable.

dv_drop_min: (**hynet_float_**) Dual variable or KKT multiplier of the voltage drop lower bound constraint or `numpy.nan` if unavailable.

dv_drop_max: (**hynet_float_**) Dual variable or KKT multiplier of the voltage drop upper bound constraint or `numpy.nan` if unavailable.

- **converter** (*pandas.DataFrame*) – Unavailable if the result is empty and, otherwise, a data frame with the converter result data, indexed by the *converter ID*, which comprises the following columns:

  p_src: (**hynet_float_**) Active power flow in MW at the source bus *into the converter*.

  p_dst: (**hynet_float_**) Active power flow in MW at the destination bus *into the converter*.

  q_src: (**hynet_float_**) Reactive power injection in Mvar at the source bus *into the grid*.

  q_dst: (**hynet_float_**) Reactive power injection in Mvar at the destination bus *into the grid*.

  loss_err: (**hynet_float_**) Loss error in MW due to noncomplementary modes of the converter.

  loss_err_pre: (**hynet_float_**) Only available if the QCQP was pre-solved to detect and fix the converter modes. Loss error in MW in the pre-solution due to noncomplementary modes of the converter.

  dv_p_fwd_min: (**hynet_float_**) Dual variable or KKT multiplier of the lower bound on the converter's forward mode active power flow or `numpy.nan` if unavailable.

  dv_p_fwd_max: (**hynet_float_**) Dual variable or KKT multiplier of the upper bound on the converter's forward mode active power flow or `numpy.nan` if unavailable.

  dv_p_bwd_min: (**hynet_float_**) Dual variable or KKT multiplier of the lower bound on the converter's backward mode active power flow or `numpy.nan` if unavailable.

  dv_p_bwd_max: (**hynet_float_**) Dual variable or KKT multiplier of the upper bound on the converter's backward mode active power flow or `numpy.nan` if unavailable.

  dv_cap_src_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region at the source bus or `numpy.nan` if unavailable.

  dv_cap_src_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region at the source bus or `numpy.nan` if unavailable.

dv_cap_dst_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region at the destination bus or `numpy.nan` if unavailable.

dv_cap_dst_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region at the destination bus or `numpy.nan` if unavailable.

dv_cap_src_lt: (**hynet_float_**) Dual variable or KKT multiplier of the left-top half-space of of the capability region at the source bus or `numpy.nan` if unavailable.

dv_cap_src_rt: (**hynet_float_**) Dual variable or KKT multiplier of the right-top half-space of of the capability region at the source bus or `numpy.nan` if unavailable.

dv_cap_src_lb: (**hynet_float_**) Dual variable or KKT multiplier of the left-bottom half-space of of the capability region at the source bus or `numpy.nan` if unavailable.

dv_cap_src_rb: (**hynet_float_**) Dual variable or KKT multiplier of the right-bottom half-space of of the capability region at the source bus or `numpy.nan` if unavailable.

dv_cap_dst_lt: (**hynet_float_**) Dual variable or KKT multiplier of the left-top half-space of of the capability region at the destination bus or `numpy.nan` if unavailable.

dv_cap_dst_rt: (**hynet_float_**) Dual variable or KKT multiplier of the right-top half-space of of the capability region at the destination bus or `numpy.nan` if unavailable.

dv_cap_dst_lb: (**hynet_float_**) Dual variable or KKT multiplier of the left-bottom half-space of of the capability region at the destination bus or `numpy.nan` if unavailable.

dv_cap_dst_rb: (**hynet_float_**) Dual variable or KKT multiplier of the right-bottom half-space of of the capability region at the destination bus or `numpy.nan` if unavailable.

- **injector** (*pandas.DataFrame*) – Unavailable if the result is empty and, otherwise, a data frame with the injector result data, indexed by the *injector ID*, which comprises the following columns:

  s: (**hynet_complex_**) Apparent power injection in MVA.

  cost_p: (**hynet_float_**) Cost of the active power injection in dollars or `numpy.nan` if no cost function was provided.

  cost_q: (**hynet_float_**) Cost of the reactive power injection in dollars or `numpy.nan` if no cost function was provided.

  dv_cap_p_min: (**hynet_float_**) Dual variable or KKT multiplier of the active power lower bound of the capability region or `numpy.nan` if unavailable.

  dv_cap_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region or `numpy.nan` if unavailable.

  dv_cap_p_max: (**hynet_float_**) Dual variable or KKT multiplier of the active power upper bound of the capability region or `numpy.nan` if unavailable.

  dv_cap_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region or `numpy.nan` if unavailable.

  dv_cap_lt: (**hynet_float_**) Dual variable or KKT multiplier of the left-top half-space of the capability region or `numpy.nan` if unavailable.

dv_cap_rt: (**hynet_float_**) Dual variable or KKT multiplier of the right-top half-space of the capability region or `numpy.nan` if unavailable.

dv_cap_lb: (**hynet_float_**) Dual variable or KKT multiplier of the left-bottom half-space of the capability region or `numpy.nan` if unavailable.

dv_cap_rb: (**hynet_float_**) Dual variable or KKT multiplier of the right-bottom half-space of the capability region or `numpy.nan` if unavailable.

References

original_scenario
Return the original scenario data.

scenario
Return the scenario with the identified *maximum load*.

3.1.4.5 Module contents

Maximum loadability functionality in *hynet*.

3.1.5 hynet.opf package

3.1.5.1 Submodules

3.1.5.2 hynet.opf.calc module

Calculation of the optimal power flow.

`hynet.opf.calc.`**calc_opf**(*data*, *scenario_id=0*, *solver=None*, *solver_type=<SolverType.QCQP: 'QCQP'>*, *initial_point_generator=None*, *converter_loss_error_tolerance=0.0005*)

Calculate the optimal power flow.

This function formulates and solves the optimal power flow (OPF) problem. The solver or solver type may be specified explicitly, otherwise an appropriate solver is selected automatically.

Parameters

- **data** (`DBConnection or Scenario or OPFModel`) – Connection to a *hynet* grid database, a `Scenario` object, or a `OPFModel` object.
- **scenario_id** (*hynet_id_, optional*) – Identifier of the scenario. This argument is ignored if `data` is a `Scenario` or `OPFModel` object.
- **solver** (`SolverInterface, optional`) – Solver for the QCQP problem; the default automatically selects an appropriate solver of the specified solver type.
- **solver_type** (`SolverType, optional`) – Solver type for the automatic solver selection (default `SolverType.QCQP`). It is ignored if `solver` is not `None`.
- **initial_point_generator** (`InitialPointGenerator or None, optional`) – Initial point generator for QCQP solvers (ignored for relaxation-based solvers). By default, an appropriate initial point generator is selected if a computationally efficient SOCR solver is installed. Set to `None` to skip the initial point generation.
- **converter_loss_error_tolerance** (*hynet_float_, optional*) – Tolerance for the converter loss error in MW (default `5e-4`). If `None`, the OPF model's (default) setting is retained.

Returns **result** – Optimal power flow solution.

Return type *OPFResult*

See also:

hynet.scenario.representation.Scenario(), *hynet.opf.result.OPFResult()*, *hynet.reduction.copper_plate.reduce_to_copper_plate()*

hynet.opf.initial_point() Initial point generators.

hynet.opf.calc.**get_default_initial_point_generator**()
Return the default initial point generator for the current system.

If a sufficiently efficient SOCR solver is available, the utilization of a copper plate based initial point for the solution of the nonconvex QCQP typically improves the overall performance, i.e., that the reduced number of iterations of the QCQP solver outweighs the computational cost for the initial point.

3.1.5.3 hynet.opf.initial_point module

Initial point generation for the QCQP solvers for the OPF problem.

class hynet.opf.initial_point.**CopperPlateInitialPointGenerator**(*solver*)
Bases: *hynet.system.initial_point.InitialPointGenerator*

Copper plate based initial point generator for the QCQP OPF solvers.

This generator returns an initial point for the solution of the OPF QCQP that comprises the optimal dispatch of the copper plate reduction of the model. The copper plate solution is typically fast to compute and can reduce the number of iterations required to solve the nonconvex QCQP, i.e., it can improve the overall performance when solving the nonconvex problem.

solver
Return the solver for the initial point computation.

3.1.5.4 hynet.opf.model module

Optimal power flow model of *hynet*.

class hynet.opf.model.**OPFModel**(*scenario*, *verify_scenario=True*)
Bases: *hynet.system.model.SystemModel*

Optimal power flow model for a steady-state scenario of a grid.

Based on the specification of a scenario via a `Scenario` object, this class provides the methods to generate a quadratically constrained quadratic program (QCQP) that captures the optimal power flow problem.

See also:

hynet.scenario.representation.Scenario Specification of a steady-state grid scenario.

hynet.opf.calc.calc_opf Calculate an optimal power flow.

create_result(*qcqp_result*, *total_time=nan*, *qcqp_result_pre=None*)
Create and return an OPF result object.

Parameters

- **qcqp_result** (*hynet.qcqp.result.QCQPResult*) – Solution of the OPF QCQP.
- **total_time** (*hynet_float_, optional*) – Total time for solving the OPF, cf. `hynet.opf.calc.calc_opf`.
- **qcqp_result_pre** (*QCQPResult, optional*) – Pre-solution of the OPF QCQP for converter mode detection.

Returns result

Return type *hynet.opf.result.OPFResult*

hynet.opf.model.**select_cost_scaling**(*scenario*)
Return a suitable scaling factor for the objective of the OPF problem.

3.1.5.5 hynet.opf.result module

Representation of an optimal power flow result.

class hynet.opf.result.**OPFResult**(*model*, *qcqp_result*, *total_time=nan*, *qcqp_result_pre=None*)
Bases: *hynet.system.result.SystemResult*

Result of an optimal power flow calculation.

Remark: In the data frames below, the respective column for the dual variables of a type of constraint (e.g., voltage drop) is only present if at least one constraint of this constraint type appears in the problem formulation.

Parameters

- **model** (OPFModel) – Model for the processed optimal power flow problem.
- **empty** (*bool*) – True if the object does not contain any result data and False otherwise.
- **solver** (SolverInterface) – Solver object by which the result was obtained.
- **solver_status** (SolverStatus) – Status reported by the solver.
- **solver_time** (*float*) – Duration of the call to the solver in seconds.
- **optimal_value** (*float*) – Optimal objective value or numpy.nan if the solver failed.
- **total_time** (*float or numpy.nan*) – Total time for the OPF calculation, including the modeling, solving, and result assembly. If not provided, this time is set to numpy.nan.
- **reconstruction_mse** (*float*) – Unavailable if the result is empty and, otherwise, the mean squared error of the reconstructed bus voltages in case of a relaxation and numpy.nan otherwise.
- **bus** (*pandas.DataFrame, optional*) – Unavailable if the result is empty and, otherwise, a data frame with the bus result data, indexed by the *bus ID*, which comprises the following columns:

  v: (hynet_complex_) Bus voltage rms phasor (AC) or bus voltage magnitude (DC).

  s_shunt: (hynet_complex_) Shunt apparent power in MVA. The real part constitutes the shunt losses in MW and the *negated* imaginary part constitutes the reactive power *injection*.

  bal_err: (hynet_complex_) Power balance residual in MVA, i.e., the evaluation of the complex-valued power balance equation at the system state. Theoretically, this should be identical to zero, but due to a limited solver accuracy and/or inexactness of the relaxation it is only approximately zero. This residual supports the assessment of solution accuracy and validity.

  dv_bal_p: (hynet_float_) Dual variable or KKT multiplier of the active power balance constraint in $/MW. In case of exactness of the relaxation (or zero duality gap in the QCQP), these dual variables equal the locational marginal prices (LMPs) for active power, cf. [1].

  dv_bal_q: (hynet_float_) Dual variable or KKT multiplier of the reactive power balance constraint in $/Mvar. In case of exactness of the relaxation (or zero

duality gap in the QCQP), these dual variables equal the locational marginal prices (LMPs) for reactive power, cf. [1].

dv_v_min: (**hynet_float_**) Dual variable or KKT multiplier of the voltage lower bound in $/p.u..

dv_v_max: (**hynet_float_**) Dual variable or KKT multiplier of the voltage upper bound in $/p.u..

- **branch** (*pandas.DataFrame, optional*) – Unavailable if the result is empty and, otherwise, a data frame with the branch result data, indexed by the *branch ID*, which comprises the following columns:

  s_src: (**hynet_complex_**) Apparent power flow in MVA at the source bus (measured as a flow *into* the branch).

  s_dst: (**hynet_complex_**) Apparent power flow in MVA at the destination bus (measured as a flow *into* the branch).

  i_src: (**hynet_complex_**) Current flow in p.u. at the source bus (measured as a flow *into* the branch).

  i_dst: (**hynet_complex_**) Current flow in p.u. at the destination bus (measured as a flow *into* the branch).

  v_drop: (**hynet_float_**) Relative voltage magnitude drop from the source bus to the destination bus.

  angle_diff: (**hynet_float_**) Bus voltage angle difference in degrees between the source and destination bus.

  effective_rating: (**hynet_float_**) Ampacity in terms of a long-term MVA rating at the *actual* bus voltage. If no rating is available, it is set to `numpy.nan`.

  rel_err: (**hynet_float_**) Branch-related relative reconstruction error $\kappa_k(V^\star)$ as defined in equation (24) in [1] in case of a relaxed OPF or `numpy.nan` otherwise.

  dv_i_max_src: (**hynet_float_**) Dual variable or KKT multiplier of the ampacity constraint at the source bus in $/p.u. or `numpy.nan` if unavailable.

  dv_i_max_dst: (**hynet_float_**) Dual variable or KKT multiplier of the ampacity constraint at the destination bus in $/p.u. or `numpy.nan` if unavailable.

  dv_angle_min: (**hynet_float_**) Dual variable or KKT multiplier of the angle difference lower bound constraint or `numpy.nan` if unavailable.

  dv_angle_max: (**hynet_float_**) Dual variable or KKT multiplier of the angle difference upper bound constraint or `numpy.nan` if unavailable.

  dv_real_part: (**hynet_float_**) Dual variable or KKT multiplier of the +/-90 degrees constraint on the angle difference (cf. equation (27) in [2]) or `numpy.nan` if unavailable.

  dv_drop_min: (**hynet_float_**) Dual variable or KKT multiplier of the voltage drop lower bound constraint or `numpy.nan` if unavailable.

  dv_drop_max: (**hynet_float_**) Dual variable or KKT multiplier of the voltage drop upper bound constraint or `numpy.nan` if unavailable.

- **converter** (*pandas.DataFrame*) – Unavailable if the result is empty and, otherwise, a data frame with the converter result data, indexed by the *converter ID*, which comprises the following columns:

  p_src: (**hynet_complex_**) Active power flow in MW at the source bus *into the converter*.

  p_dst: (**hynet_complex_**) Active power flow in MW at the destination bus *into the converter*.

q_src: (**hynet_complex_**) Reactive power injection in Mvar at the source bus *into the grid*.

q_dst: (**hynet_complex_**) Reactive power injection in Mvar at the destination bus *into the grid*.

loss_err: (**hynet_float_**) Loss error in MW due to noncomplementary modes of the converter.

loss_err_pre: (**hynet_float_**) Only available if the OPF QCQP was pre-solved to detect and fix the converter modes. Loss error in MW in the pre-solution due to noncomplementary modes of the converter.

dv_p_fwd_min: (**hynet_float_**) Dual variable or KKT multiplier of the lower bound on the converter's forward mode active power flow in $/MW or `numpy.nan` if unavailable.

dv_p_fwd_max: (**hynet_float_**) Dual variable or KKT multiplier of the upper bound on the converter's forward mode active power flow in $/MW or `numpy.nan` if unavailable.

dv_p_bwd_min: (**hynet_float_**) Dual variable or KKT multiplier of the lower bound on the converter's backward mode active power flow in $/MW or `numpy.nan` if unavailable.

dv_p_bwd_max: (**hynet_float_**) Dual variable or KKT multiplier of the upper bound on the converter's backward mode active power flow in $/MW or `numpy.nan` if unavailable.

dv_cap_src_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region at the source bus in $/Mvar or `numpy.nan` if unavailable.

dv_cap_src_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region at the source bus in $/Mvar or `numpy.nan` if unavailable.

dv_cap_dst_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region at the destination bus in $/Mvar or `numpy.nan` if unavailable.

dv_cap_dst_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region at the destination bus in $/Mvar or `numpy.nan` if unavailable.

dv_cap_src_lt: (**hynet_float_**) Dual variable or KKT multiplier of the left-top half-space of of the capability region at the source bus in $/MVA or `numpy.nan` if unavailable.

dv_cap_src_rt: (**hynet_float_**) Dual variable or KKT multiplier of the right-top half-space of of the capability region at the source bus in $/MVA or `numpy.nan` if unavailable.

dv_cap_src_lb: (**hynet_float_**) Dual variable or KKT multiplier of the left-bottom half-space of of the capability region at the source bus in $/MVA or `numpy.nan` if unavailable.

dv_cap_src_rb: (**hynet_float_**) Dual variable or KKT multiplier of the right-bottom half-space of of the capability region at the source bus in $/MVA or `numpy.nan` if unavailable.

dv_cap_dst_lt: (**hynet_float_**) Dual variable or KKT multiplier of the left-top half-space of of the capability region at the destination bus in $/MVA or `numpy.nan` if unavailable.

dv_cap_dst_rt: (**hynet_float_**) Dual variable or KKT multiplier of the right-top half-space of of the capability region at the destination bus in $/MVA or `numpy.nan` if unavailable.

dv_cap_dst_lb: (**hynet_float_**) Dual variable or KKT multiplier of the left-bottom half-space of of the capability region at the destination bus in $/MVA or `numpy.nan` if unavailable.

dv_cap_dst_rb: (**hynet_float_**) Dual variable or KKT multiplier of the right-bottom half-space of of the capability region at the destination bus in $/MVA or `numpy.nan` if unavailable.

- **injector** (*pandas.DataFrame*) – Unavailable if the result is empty and, otherwise, a data frame with the injector result data, indexed by the *injector ID*, which comprises the following columns:

  s: (**hynet_complex_**) Apparent power injection in MVA.

  cost_p: (**hynet_float_**) Cost of the active power injection in dollars or `numpy.nan` if no cost function was provided.

  cost_q: (**hynet_float_**) Cost of the reactive power injection in dollars or `numpy.nan` if no cost function was provided.

  dv_cap_p_min: (**hynet_float_**) Dual variable or KKT multiplier of the active power lower bound of the capability region in $/MW or `numpy.nan` if unavailable.

  dv_cap_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region in $/Mvar or `numpy.nan` if unavailable.

  dv_cap_p_max: (**hynet_float_**) Dual variable or KKT multiplier of the active power upper bound of the capability region in $/MW or `numpy.nan` if unavailable.

  dv_cap_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region in $/Mvar or `numpy.nan` if unavailable.

  dv_cap_lt: (**hynet_float_**) Dual variable or KKT multiplier of the left-top half-space of the capability region in $/MVA or `numpy.nan` if unavailable.

  dv_cap_rt: (**hynet_float_**) Dual variable or KKT multiplier of the right-top half-space of the capability region in $/MVA or `numpy.nan` if unavailable.

  dv_cap_lb: (**hynet_float_**) Dual variable or KKT multiplier of the left-bottom half-space of the capability region in $/MVA or `numpy.nan` if unavailable.

  dv_cap_rb: (**hynet_float_**) Dual variable or KKT multiplier of the right-bottom half-space of the capability region in $/MVA or `numpy.nan` if unavailable.

References

3.1.5.6 hynet.opf.visual module

Visualization functionality for optimal power flow results.

`hynet.opf.visual.`**`show_ampacity_dual_profile`**(*result*, *id_label=False*)

Show the ampacity dual variable profile for an OPF result.

For a consistent appearance in case of nonconsecutive or custom IDs, **the branches are numbered consecutively starting from 1 for the labeling of the x-axis**. To enforce labeling of the ticks with the IDs, set `id_label=True`.

Parameters

- **result** (*OPFResult*) – Optimal power flow result that shall be visualized.
- **id_label** (*bool, optional*) – If `True`, the linear ticks are relabeled with the branch IDs.

Returns fig

Return type matplotlib.figure.Figure

hynet.opf.visual.**show_branch_flow_profile**(*result*, *id_label=False*)

Show the branch flow profile for an OPF result.

For a consistent appearance in case of nonconsecutive or custom IDs, **the branches are numbered consecutively starting from 1 for the labeling of the x-axis**. To enforce labeling of the ticks with the IDs, set `id_label=True`.

Parameters

- **result** (OPFResult) – Optimal power flow result that shall be visualized.
- **id_label** (*bool, optional*) – If `True`, the linear ticks are relabeled with the branch IDs.

Returns fig

Return type matplotlib.figure.Figure

hynet.opf.visual.**show_branch_reconstruction_error**(*result*, *show_s_bal_err=True*, *show_p_bal_err=False*, *show_q_bal_err=False*, *show_p_dual=True*, *show_q_dual=True*, *show_z_bar_abs=False*, *show_z_bar_real=False*, *show_z_bar_imag=False*)

Show the branch-related reconstruction error for an OPF result.

If the SOCR of an OPF problem is solved and the graph traversal based rank-1 approximation is used, the branch-related "contributions" to the reconstruction error (and, if present, inexactness of the relaxation) are characterized by $\kappa_k(V^\star)$ defined in equation (24) in [1]. Especially if the scenario features the *hybrid architecture* and inexactness occurred, this visualization may assist in finding the cause of the pathological price profile (see also [1], Section VII, for more details).

Parameters

- **result** (OPFResult) – Optimal power flow result that shall be visualized.
- **show_s_bal_err** (*bool, optional*) – If `True` (default), the maximum apparent power balance error magnitude at the adjacent buses of the branches is shown as well.
- **show_p_bal_err** (*bool, optional*) – If `True` (default `False`), the maximum active power balance error modulus at the adjacent buses of the branches is shown as well.
- **show_q_bal_err** (*bool, optional*) – If `True` (default `False`), the maximum reactive power balance error modulus at the adjacent buses of the branches is shown as well.
- **show_p_dual** (*bool, optional*) – If `True` (default), the minimum active power balance dual variable at the adjacent buses of the branches is shown as well.
- **show_q_dual** (*bool, optional*) – If `True` (default), the minimum reactive power balance dual variable at the adjacent buses of the branches is shown as well.
- **show_z_bar_abs** (*bool, optional*) – If `True` (default `False`), the modulus of the branch series impedance is shown as well.
- **show_z_bar_real** (*bool, optional*) – If `True` (default `False`), the modulus of the branch series resistance is shown as well.
- **show_z_bar_imag** (*bool, optional*) – If `True` (default `False`), the modulus of the branch series reactance is shown as well.

Returns fig

Return type matplotlib.figure.Figure

See also:

hynet.opf.visual.show_power_balance_error(), *hynet.scenario.representation.Scenario.verify_hybrid_architecture_conditions()*, *hynet.qcqp.rank1approx.GraphTraversalRank1Approximator()*

References

hynet.opf.visual.**show_converter_flow_profile**(*result*, *id_label=False*)
Show the converter active power flow profile for an OPF result.

For a consistent appearance in case of nonconsecutive or custom IDs, **the converters are numbered consecutively starting from 1 for the labeling of the x-axis**. To enforce labeling of the ticks with the IDs, set `id_label=True`.

Parameters

- **result** (*OPFResult*) – Optimal power flow result that shall be visualized.
- **id_label** (*bool, optional*) – If `True`, the linear ticks are relabeled with the converter IDs.

Returns fig

Return type matplotlib.figure.Figure

hynet.opf.visual.**show_dispatch_profile**(*result*, *id_label=False*)
Show the active power injector dispatch profile for an OPF result.

For a consistent appearance in case of nonconsecutive or custom IDs, **the injectors are numbered consecutively starting from 1 for the labeling of the x-axis**. To enforce labeling of the ticks with the IDs, set `id_label=True`.

Parameters

- **result** (*OPFResult*) – Optimal power flow result that shall be visualized.
- **id_label** (*bool, optional*) – If `True`, the linear ticks are relabeled with the injector IDs.

Returns fig

Return type matplotlib.figure.Figure

hynet.opf.visual.**show_lmp_profile**(*result*, *id_label=False*)
Show the LMP profile or power balance dual variables for an OPF result.

For a consistent appearance in case of nonconsecutive or custom IDs, **the buses are numbered consecutively starting from 1 for the labeling of the x-axis**. To enforce labeling of the ticks with the IDs, set `id_label=True`.

Parameters

- **result** (*OPFResult*) – Optimal power flow result that shall be visualized.
- **id_label** (*bool, optional*) – If `True`, the linear ticks are relabeled with the bus IDs.

Returns fig

Return type matplotlib.figure.Figure

hynet.opf.visual.**show_power_balance_error**(*result*, *show_duals=True*, *split_acdc=True*, *id_label=False*)
Show the power balance error at all AC and DC buses for an OPF result.

Parameters

- **result** (OPFResult) – Optimal power flow result that shall be visualized.
- **show_duals** (*bool, optional*) – If True (default), the active and reactive power balance dual variables are shown as well.
- **split_acdc** (*bool, optional*) – If True (default), the results are shown separate plots for AC and DC buses. In this case, no x-axis labels are provided. Otherwise, all buses are shown in one plot and, for the x-axis labels, the buses are numbered consecutively starting from 1.
- **id_label** (*bool, optional*) – If True (default False), the linear x-axis ticks are relabeled with the bus IDs.

Returns fig

Return type matplotlib.figure.Figure

See also:

hynet.opf.visual.show_branch_reconstruction_error(), *hynet.scenario.representation.Scenario.verify_hybrid_architecture_conditions()*

hynet.opf.visual.**show_voltage_profile**(*result*, *id_label=False*)

Show the voltage profile for an OPF result.

For a consistent appearance in case of nonconsecutive or custom IDs, **the buses are numbered consecutively starting from 1 for the labeling of the x-axis**. To enforce labeling of the ticks with the IDs, set id_label=True.

Parameters

- **result** (OPFResult) – Optimal power flow result that shall be visualized.
- **id_label** (*bool, optional*) – If True, the linear ticks are relabeled with the bus IDs.

Returns fig

Return type matplotlib.figure.Figure

3.1.5.7 Module contents

Optimal power flow functionality in *hynet*.

3.1.6 hynet.qcqp package

3.1.6.1 Submodules

3.1.6.2 hynet.qcqp.problem module

Representation of a *hynet*-specific QCQP.

class hynet.qcqp.problem.**Constraint**

Bases: hynet.qcqp.problem.ConstraintBase

Specifies a constraint v^H C v + c^T f + a^T s + r^T z <=/== b.

Parameters

- **name** (*str*) – Name of the constraint and column name for the result table generation in QCQPResult.

- **table** (*str or None*) – Name of the table with which this constraint is associated. This is used to generate the result tables in `QCQPResult`. Set to `None` to ignore the associated values during the result table generation.
- **id** (*hynet_id_*) – Identifier of the row with which this constraint is associated. This is used to generate the result tables in `QCQPResult`.
- **C** (*hynet_sparse_ or None*) – Hermitian matrix `C` of the constraint function or `None` if this term is omitted.
- **c** (*hynet_sparse_ or None*) – Vector `c` of the constraint function or `None` if this term is omitted.
- **a** (*hynet_sparse_ or None*) – Vector `a` of the constraint function or `None` if this term is omitted.
- **r** (*hynet_sparse_ or None*) – Vector `r` of the constraint function or `None` if this term is omitted.
- **b** (*hynet_float_*) – Right-hand side scalar `b` of the constraint.
- **scaling** (*hynet_float_*) – Scaling factor of the constraint.

evaluate(*p*)

Return `y = v^H C v + c^T f + a^T s + r^T z - b`.

Parameters **p** (`QCQPPoint`) – Point that specifies `v`, `f`, `s`, and `z`.

Returns **y** – Constraint value `y` at the given point `p`.

Return type hynet_float_

class hynet.qcqp.problem.**ObjectiveFunction**(*C*, *c*, *a*, *r*, *scaling*)

Bases: `object`

Specifies an objective `g(v,f,s,z) = v^H C v + c^T f + a^T s + r^T z`.

Parameters

- **C** (*hynet_sparse_*) – Hermitian matrix `C` of the objective function.
- **c** (*hynet_sparse_*) – Vector `c` of the objective function.
- **a** (*hynet_sparse_*) – Vector `a` of the objective function.
- **r** (*hynet_sparse_*) – Vector `r` of the objective function.
- **scaling** (*hynet_float_*) – Scaling factor for the objective.

evaluate(*p*)

Evaluate the function, i.e., return `y = g(v, f, s, z)`.

Parameters **p** (`QCQPPoint`) – Point that specifies `v`, `f`, `s`, and `z`.

Returns **y** – Function value `y` at the given point `p`.

Return type hynet_float_

class hynet.qcqp.problem.**QCQP**(*obj_func*, *eq_crt*, *ineq_crt*, *lb*, *ub*, *edges*, *roots*, *normalization=None*, *initial_point=None*, *use_buffered_vectorization=True*)

Bases: `object`

Specification of a quadratically constrained quadratic program:

```
  min   v^H C v + c^T f + a^T s + r^T z
v,f,s,z

  s.t.  v^H C'_i v + c'_i^T f + a'_i^T s + r'_i^T z == b'_i, i = 1,...,N
        v^H C"_j v + c"_j^T f + a"_j^T s + r"_j^T z <= b"_j, j = 1,...,M
        v_lb <= |v| <= v_ub
```

(continues on next page)

(continued from previous page)

```
f_lb <=  f  <= f_ub
s_lb <=  s  <= s_ub
z_lb <=  z  <= z_ub
```

The vector `v` is complex-valued, while `f`, `s`, and `z` are real-valued. The bounds on `|v|` are optional to implement by the solver. They are assumed to be captured by the inequality constraints, while `v_lb` and `v_ub` may be used to support convergence of the solver. In contrary, the bounds on `f`, `s`, and `z` are mandatory to implement. A bound is ignored if the respective element is set to `numpy.nan`.

The objective function, constraint functions, and optimization variables may be scaled to improve the numerical conditioning of the problem. All coefficient matrices, coefficient vectors, bounds as well as the initial point, if provided, must be specified for the *scaled* problem. The scaling factors for the objective and the constraints can be specified via their `scaling` attribute, while the scaling of the optimization variables is defined via `normalization` (see below). This scaling information is utilized in `QCQPResult` to adjust the optimizer, the optimal value, and the dual variables to represent the solution of the unscaled problem.

Caution: If the QCQP object is used in repeated computations with modifications of the constraints, the constraint vectorization buffering must be deactivated, see `use_buffered_vectorization`.

Parameters

- **obj_func** (`ObjectiveFunction`) – Specification of the objective function.
- **eq_crt** (*numpy.ndarray[Constraint]*) – Specification of the equality constraints.
- **ineq_crt** (*numpy.ndarray[Constraint]*) – Specification of the inequality constraints.
- **lb** (`QCQPPoint`) – Specification of the lower bounds on `|v|`, `f`, `s`, and `z`.
- **ub** (`QCQPPoint`) – Specification of the upper bounds on `|v|`, `f`, `s`, and `z`.
- **edges** (*(numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_])*) – Specification of the sparsity pattern of the lower-triangular part of the constraint matrices via the edges `(edge_src, edge_dst)` of an undirected graph. The constraint matrices may only have nonzero entries in row/column `edge_src[i]` and column/row `edge_dst[i]`, with `i in range(K)` where `K = len(edge_src) = len(edge_dst)`, as well as on the diagonal. The edges specified in `edges` must be *unique*, i.e., there must not exist any parallel or anti-parallel edges, and must refer to the *lower-triangular part*, i.e., `edges[0] > edges[1]`. If case of any doubts, call `hynet.utilities.graph.eliminate_parallel_edges` before assigning.
- **roots** (*numpy.ndarray[hynet_int_]*) – Root nodes of the undirected graph defined by `edges` that specifies the sparsity pattern. At these root nodes the absolute angle of the optimal `v` is set to zero. (Note that, due to the quadratic form in `v`, the optimal objective value is invariant w.r.t. an absolute phase shift within a connected component of the sparsity-defining graph.)
- **normalization** (`QCQPPoint`) – The attributes `v`, `f`, `s`, and `z` specify the scaling of the corresponding optimization variable. This scaling is considered in the creation of the QCQP result, where the optimal value as well as the dual variables of the box constraints are adjusted accordingly.
- **initial_point** (`QCQPPoint` *or None*) – Initial point for the solver or None if omitted. (The initial point is typically only utilized in solvers for the nonconvex QCQP, but ignored in SDR and SOCR solvers.)
- **use_buffered_vectorization** (*bool*) – If True (default), the constraint vectorization is buffered to avoid its repeated computation. This QCQP object supports

a vectorized representation, which is primarily intended for solver interfaces of relaxations, cf. `get_vectorization`. As the vectorization of the constraints is computationally demanding, the result is buffered and reused if called repeatedly. If the *constraints are modified* between the calls, the buffering must be deactivated to update the vectorization.

dim_f
Return the dimension of the ambient space of `f`.

dim_s
Return the dimension of the ambient space of `s`.

dim_v
Return the dimension of the ambient space of `v`.

dim_z
Return the dimension of the ambient space of `z`.

get_constraint_vectorization()
Return a vectorized representation of the constraints.

This class supports a vectorized representation of the QCQP, see `get_vectorization` for more details. This method returns the parameters for the vectorization of the equality and inequality constraints, i.e., `Ax == b` and `Bx <= d`.

See also:

hynet.qcqp.problem.QCQP.get_vectorization()

Returns **A, b, B, d** – Vectorization of the equality and inequality constraints.

Return type tuple

get_vectorization()
Return a vectorized representation of this QCQP:

```
min    c^T x    s.t.    Ax == b,   Bx <= d,   x_lb <= x <= x_ub
 x
```

This representation is primarily intended for solver interfaces to *relaxations* of the QCQP and based on a reformulation in three steps, i.e.,

1) The complex-valued optimization variable `v` is replaced by a matrix `V`:

```
v^H C v = tr(v^H C v) = tr(Cvv^H) = tr(CV) with V = vv^H
```

For equivalence to the original problem, `V` must be Hermitian (`V = V^H`), positive semidefinite (`V >= 0`) and have rank 1 (`rank(V) = 1`). **Note that this vectorization does not take care of this, it only reformulates the problem in in terms of ‘‘V‘‘.** (For example, the semidefinite relaxation will include the psd constraint but omit the rank constraint.)

2) The sparsity of the constraint matrices is utilized to reduce the number of effective variables and vectorize the matrices, i.e.,

```
tr(CV) = c_vec^T v_tld
```

where `c_vec = self._mtx2vec(C)` and

```
v_tld = [V_{1,1}                          ^
         ...                              | N = self.dim_v
         V_{N,N}                          v
         real(V_{e_src(1),e_dst(1)}       ^
         ...                              | K = self.num_edges
         real(V_{e_src(K),e_dst(K)}       v
```

(continues on next page)

(continued from previous page)

```
        imag(V_{e_src(1),e_dst(1)}     ^
        ...                            | K = self.num_edges
        imag(V_{e_src(K),e_dst(K)}]    v
```

3) All optimization variables are stacked, i.e.,

```
x = [v_tld^T, f^T, s^T, z^T]^T
```

and the objective as well as the equality, inequality, and box constraints are adapted accordingly.

Caution: This representation is **only reasonable with additional constraints** on the `v_tld`-part of `x`. For example, psd constraints on principal submatrices for the SOC relaxation or a psd constraint on the "reassembled" matrix `V` for the SDR.

Returns **(c, A, b, B, d, x_lb, x_ub)** – Parameters of the vectorized QCQP representation.

Return type tuple

num_edges
Return the number of edges in the sparsity-defining graph.

split_vectorization_bound_dual(*dual_var*)
Return the dual variable of a bound on `x` as a QCQP point.

Splits the dual variable of a bound on the optimization variable `x` of the vectorized problem (`x >= x_lb` or `x <= x_ub`) into the dual variables of the respective bound on `f`, `s`, and `z` and returns them as a QCQP point (with `v` set to `None` as the respective magnitude bounds are optional).

Parameters **dual_var** (*numpy.ndarray[hynet_float_]*) – Dual variable of a bound on `x` of the vectorized problem.

Returns **point** – QCQP point with the dual variables of the respective bounds on `|v|`, `f`, `s`, and `z`.

Return type *QCQPPoint*

split_vectorization_optimizer(*x*)
Return `(V, f, s, z)` for the optimizer `x` of the vectorized problem.

The partitioning in `x` is assumed as defined in `get_vectorization`. The vectorized representation `v_tld` of `V` is retracted, i.e., `V` is populated with the diagonal and off-diagonal elements specified by `v_tld`.

class hynet.qcqp.problem.**QCQPPoint**(*v*, *f*, *s*, *z*)
Bases: `object`

Point in the space of the optimization variables of the QCQP problem.

Parameters

- **v** (*numpy.ndarray[hynet_complex_] or numpy.ndarray[hynet_float_]*) –
- **f** (*numpy.ndarray[hynet_float_]*) –
- **s** (*numpy.ndarray[hynet_float_]*) –
- **z** (*numpy.ndarray[hynet_float_]*) –

copy()
Return a deep copy of this point.

reciprocal()
Set the point to the element-wise reciprocal of the variables.

scale(*scaling*)
Scale the point by `scaling`.

Parameters **scaling** (*QCQPPoint or hynet_float_*) – Scaling factor, which can either be a numeric value, to scale all variables equally, or a QCQP point, to scale the variables individually, where the attributes `v`, `f`, `s`, and `z` specify the scaling factor for the corresponding variable.

3.1.6.3 hynet.qcqp.rank1approx module

Rank-1 approximation of partial Hermitian matrices.

This module provides an object-oriented representation of the rank-1 approximation functionality provided in *hynet*'s utilities.

See also:

hynet.utilities.rank1approx, *hynet.qcqp.solver*

class hynet.qcqp.rank1approx.**GraphTraversalRank1Approximator**
Bases: *hynet.qcqp.rank1approx.Rank1Approximator*

Graph traversal based rank-1 approximation of partial Hermitian matrices.

Please refer to `rank1approx_via_traversal` for more details.

See also:

hynet.utilities.rank1approx.rank1approx_via_traversal

class hynet.qcqp.rank1approx.**LeastSquaresRank1Approximator**(*grad_thres=1e-07*, *mse_rel_thres=0.002*, *max_iter=300*, *show_convergence_plot=False*)
Bases: *hynet.qcqp.rank1approx.Rank1Approximator*

Graph traversal based rank-1 approximation of partial Hermitian matrices.

Please refer to `rank1approx_via_least_squares` for more details.

Parameters

- **grad_thres** (*hynet_float_*) –
- **mse_rel_thres** (*hynet_float_*) –
- **max_iter** (*hynet_int_*) –
- **show_convergence_plot** (*bool*) –

See also:

hynet.utilities.rank1approx.rank1approx_via_least_squares

class hynet.qcqp.rank1approx.**Rank1Approximator**
Bases: `abc.ABC`

Abstract base class for rank-1 approximators for partial Hermitian matrices.

Derived classes implement a rank-1 approximation for partial Hermitian matrices, where the sparsity pattern of the latter is defined by its associated undirected graph.

static calc_mse(*V*, *v*, *edges*)
Calculate the mean squared error for the rank-1 approximation.

static calc_rel_err(*V*, *v*, *edges*)
Calculate the relative reconstruction error for all edges.

3.1.6.4 hynet.qcqp.result module

Representation of the result of a *hynet*-specific QCQP.

class hynet.qcqp.result.**QCQPResult**(*qcqp*, *solver*, *solver_status*, *solver_time*, *optimizer=None*, *V=None*, *optimal_value=None*, *dv_lb=None*, *dv_ub=None*, *dv_eq=None*, *dv_ineq=None*)

Bases: object

Solution of a quadratically constrained quadratic program.

Caution: The constructor adjusts the optimal objective value, the optimizer, and the dual variables according to the scaling of the problem, i.e., the properties of the result correspond to the *unscaled* QCQP.

Parameters

- **qcqp** (QCQP) – Problem specification.
- **solver** (SolverInterface) – Solver object by which the result was obtained.
- **solver_status** (SolverStatus) – Status reported by the solver after performing the optimization.
- **solver_time** (*float*) – Duration of the call to the solver in seconds.
- **optimizer** (*QCQPPoint or None*) – Optimal optimization variable `(v^*, f^*, s^*, z^*)` or `None` if the solver failed. In case of a relaxation, `v^*` is the rank-1 approximation of `V^*`.
- **V** (*hynet_sparse_ or None*) – Optimal optimization variable `V^*` in case of a relaxation or `None` if the solver failed or a nonconvex-QCQP solver was employed.
- **optimal_value** (*float*) – Optimal objective value or `numpy.nan` if the solver failed.
- **reconstruction_mse** (*float*) – The mean squared error of the reconstructed `v^*` in case of a relaxation or `numpy.nan` if the solver failed or a nonconvex-QCQP solver was employed.
- **dv_lb** (*QCQPPoint or None*) – Optimal dual variables of the lower bounds on `f`, `s`, and `z` or `None` if the solver failed. The attribute `v` is set `None` as these bounds are optional, cf. the QCQP specification.
- **dv_ub** (*QCQPPoint or None*) – Optimal dual variables of the upper bounds on `f`, `s`, and `z` or `None` if the solver failed. The attribute `v` is set `None` as these bounds are optional, cf. the QCQP specification.
- **dv_eq** (*numpy.ndarray or None*) – Optimal dual variables of the equality constraints or `None` if the solver failed.
- **dv_ineq** (*numpy.ndarray or None*) – Optimal dual variables of the inequality constraints or `None` if the solver failed.

empty
Return `True` if the QCQP result does not contain an optimizer.

get_result_tables(*tables=None*, *dual_prefix='dv_'*, *value_prefix='cv_'*)
Return a dictionary of data frames with the dual and constraint result.

According to the table and ID information of the constraint objects, a dictionary of data frames is assembled that contains the dual variables and the *equality* constraint function values with the right hand side subtracted.

3.1.6.5 hynet.qcqp.solver module

Solver interface for a *hynet*-specific QCQP.

class hynet.qcqp.solver.**SolverInterface**(*verbose=False*, *param=None*, *rank1approx=None*)

Bases: abc.ABC

Abstract base class for QCQP solvers.

Derived classes implement a solver for the *hynet*-specific quadratically constrained quadratic program specified by an object of the class QCQP. A solver may solve the nonconvex QCQP, its semidefinite relaxation (SDR), or its second-order cone relaxation (SOCR).

Parameters

- **verbose** (*bool, optional*) – If True, the logging information of the solver is printed to the standard output.
- **param** (*dict[str, object], optional*) – Dictionary of parameters ({'parameter_name': value}) to modify the solver's default settings.
- **rank1approx** (Rank1Approximator, *optional*) – Rank-1 approximator for partial Hermitian matrices. This approximator is used in relaxation-based solvers.

See also:

hynet.qcqp.rank1approx Rank-1 approximators.

static adjust_absolute_phase(*qcqp*, *v*)

Return v with absolute phase adjustment according to the root nodes.

Due to the quadratic form in v, the solution of the QCQP is ambiguous w.r.t. a common phase shift of the elements in v associated with a connected component in the sparsity pattern. This method adjusts the absolute phase such that it is zero at the root node of the respective components.

Remark: This method should be used in *QCQP solvers*. In SDR and SOCR solvers, the rank-1 approximation takes care of the absolute phase adjustment.

Parameters **v** (*numpy.ndarray[hynet_complex_]*) – Optimizer v for the solved QCQP.

Returns **v** – Optimizer v with absolute phase adjustment.

Return type numpy.ndarray[hynet_complex_]

name

Return the name of the solver.

solve(*qcqp*)

Solve the given QCQP and return a QCQPResult object.

Parameters **qcqp** (QCQP) – Specification of the quadratically-constrained quadratic problem.

Returns **result** – Solution of the QCQP.

Return type *QCQPResult*

type

Return the type of the solver as a SolverType.

3.1.6.6 Module contents

Representation of a *hynet*-specific quadratically constrained quadratic program.

3.1.7 hynet.reduction package

3.1.7.1 Subpackages

hynet.reduction.large_scale package

Submodules

hynet.reduction.large_scale.combination module

Combined network reduction strategy.

hynet.reduction.large_scale.combination.**reduce_system**(*scenario*, *solver=None*, *max_island_size=None*, *rel_impedance_thres=0.05*, *feature_depth=5*, *critical_mw_diff=None*, *max_price_diff=0.0*, *show_evaluation=False*, *return_bus_id_map=False*, *preserve_aggregation=False*)

Apply a feature- and structure-preserving network reduction.

This function applies the feature- and structure-preserving network reduction described in [1] to the scenario. It comprises a topology-based, electrical coupling-based, and market-based reduction stage, which identifies and aggregates appropriate subgrids that do not contain any features and exhibit a minor impact on the overall behavior of the system. Features are defined by a Boolean-valued column `feature` of the bus and branch data frame of the scenario. If these columns are not present upon the call of this function, the standard features described in Section III in [1] are added.

Depending on the parameterization, this function performs an extensive model analysis and several OPF computations, due to which its execution may take quite some time. To track the progress, the logging of info messages may be activated via

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO)
```

With a corresponding parameterization, the individual reduction stages can be activated or deactivated in this combined reduction strategy. However, in case the need arises, the individual reduction stages as well as their evaluation may also be performed "manually" to customize the reduction process, please refer to the "See Also" section for directions to the respective modules.

Parameters

- **scenario** (Scenario) – Scenario that shall be processed.
- **solver** (SolverInterface, *optional*) – Solver for the OPF problems. The default selects the most appropriate QCQP solver among those installed.
- **max_island_size** (*int*, *optional*) – Maximum size of an "island" in terms of its number of buses. By default, it is set to approximately 1% of the total number of buses. Set this parameter to `0` to disable the reduction of "islands", i.e., only single buses and lines of buses are reduced, or to `-1` to disable the entire topology-based reduction, i.e., the reduction of single buses, lines of buses, and "islands".
- **rel_impedance_thres** (*float*, *optional*) – Relative threshold τ w.r.t. the maximum series impedance modulus that defines a strong coupling of buses (see equation (3) in [1]). Default is `0.05` (5%). Set this parameter to `0` to disable the electrical coupling-based reduction.

- **feature_depth** (*int, optional*) – Depth ϑ for which buses in the vicinity of a critical injector are declared as features, i.e., these buses can be reached from the terminal bus of a critical injector by traversing a maximum of ϑ branches. Default is `5`. Set this parameter to `0` to disable the feature refinement.
- **critical_mw_diff** (*float, optional*) – Absolute threshold in MW on the active power dispatch difference of an injector to consider it as critical. The default is 0.02% of the total active power load.
- **max_price_diff** (*float, optional*) – Threshold δ in \$/MW on fluctuations of the nodal price (LMP) from the center of a price cluster to consider the respective buses to be part of the cluster (cf. equation (4) in [1]). Set this parameter to `0` to disable the marked-based reduction.
- **show_evaluation** (*bool, optional*) – If `True` (default is `False`), the evaluation of the individual reduction stages is visualized.
- **return_bus_id_map** (*bool, optional*) – If `True` (default is `False`), a mapping from the buses of the original system to the buses of the reduced system is returned.
- **preserve_aggregation** (*bool, optional*) – If `True` (default is `False`), the information on aggregated buses in the column `aggregation` of the bus data frame is copied to the bus annotation to preserve it when storing the reduced system to a grid database. After loading from the grid database, the column can be restored via `restore_aggregation_info`.

Returns

- **evaluation** (*pandas.DataFrame*) – Data frame with information on the extent and accuracy of the reduction in the individual reduction stages. This data frame is indexed by the name of the reduction stage, with the original system named `''`, and comprises the following columns:

  bus_reduction: (hynet_float_) Ratio of the number of reduced buses w.r.t. the total number of buses before the reduction.

  branch_reduction: (hynet_float_) Ratio of the number of reduced branches w.r.t. the total number of branches before the reduction.

  cycle_reduction: (hynet_float_) Ratio of the number of reduced cycles w.r.t. the total number of cycles before the reduction.

  error_disp: (hynet_float_) Contribution-weighted mean relative *active* power dispatch error as defined in equation (1) in [1].

  error_disp_s: (hynet_float_) Contribution-weighted mean relative *apparent* power dispatch error.

  error_flow: (hynet_float_) Contribution-weighted mean relative *active* power branch flow error as defined in equation (2) in [1].

  error_flow_s: (hynet_float_) Contribution-weighted mean relative *apparent* power branch flow error.

  opf: (hynet.OPFResult) OPF result for the respective scenario.

- **bus_id_map** (*pandas.Series*) – Only returned if `return_bus_id_map` is `True`. This series is *indexed* by the bus IDs *before* network reduction, while the *values* are the bus IDs *after* network reduction, i.e., it maps the buses of the original system to the buses in the reduced system.

See also:

hynet.reduction.large_scale.features() Specification of features.

hynet.reduction.large_scale.topology() Topology-based network reduction.

hynet.reduction.large_scale.coupling() Electrical coupling-based network reduction.

hynet.reduction.large_scale.market() Market-based network reduction.

hynet.reduction.large_scale.evaluation() Evaluation of the extent and accuracy of a reduction.

hynet.reduction.large_scale.sweep() Parameter sweeps to support the configuration process.

References

hynet.reduction.large_scale.coupling module

Electrical coupling-based subgrid selection and reduction.

hynet.reduction.large_scale.coupling.**get_critical_injector_features**(*opf_reference*, *opf_reduction*, *feature_depth*, *critical_mw_diff=None*)

Determine bus features to mitigate the reduction error at critical injectors.

It has been observed that the majority of the dispatch error induced by the electrical coupling-based reduction is often due to a comparably small number of injectors. This effect may be mitigated by declaring the buses in the vicinity of critical injectors as features, see also Section V-A in [1]. This function identifies these buses and returns their IDs.

Parameters

- **opf_reference** (*OPFResult*) – OPF result of the scenario *before* the reduction.
- **opf_reduction** (*OPFResult*) – OPF result of the scenario *after* the reduction.
- **feature_depth** (*int*) – Depth ϑ for which buses in the vicinity of a critical injector are declared as features, i.e., these buses can be reached from the terminal bus of a critical injector by traversing a maximum of ϑ branches.
- **critical_mw_diff** (*float, optional*) – Absolute threshold in MW on the active power dispatch difference of an injector to consider it as critical. The default is 0.02% of the total active power load.

Returns **critical_buses** – Array of bus IDs that should be marked as a feature.

Return type numpy.ndarray[hynet_id_]

References

hynet.reduction.large_scale.coupling.**reduce_by_coupling**(*scenario*, *rel_impedance_thres=0.05*)

Apply an electrical coupling-based network reduction to the scenario.

This function performs the electrical coupling-based network reduction described in Section IV-C in [1].

Parameters

- **scenario** (*hynet.Scenario*) – Scenario that shall be processed.
- **rel_impedance_thres** (*float, optional*) – Relative threshold τ w.r.t. the maximum series impedance modulus that defines a strong coupling of buses (see equation (3) in [1]). Default is `0.05` (5%).

Returns Number of buses that were reduced.

Return type int

References

hynet.reduction.large_scale.evaluation module

Evaluation of the network reduction accuracy.

hynet.reduction.large_scale.evaluation.**evaluate_reduction**(*opf_reference*, *opf_reduction*, *name=None*)

Return an evaluation of the extent and accuracy of the reduction.

Parameters

- **opf_reference** (OPFResult) – OPF result of the scenario *before* the reduction.
- **opf_reduction** (OPFResult) – OPF result of the scenario *after* the reduction.
- **name** (*object, optional*) – Name of the returned series. This may be set, e.g., to the name of the reduction stage or characteristic parameter value.

Returns

evaluation – Series with information on the extent and accuracy of the reduction with the following entries:

bus_reduction: (hynet_float_) Ratio of the number of reduced buses w.r.t. the total number of buses before the reduction.

branch_reduction: (hynet_float_) Ratio of the number of reduced branches w.r.t. the total number of branches before the reduction.

cycle_reduction: (hynet_float_) Ratio of the number of reduced cycles w.r.t. the total number of cycles before the reduction.

error_disp: (hynet_float_) Contribution-weighted mean relative *active* power dispatch error as defined in equation (1) in [1].

error_disp_s: (hynet_float_) Contribution-weighted mean relative *apparent* power dispatch error.

error_flow: (hynet_float_) Contribution-weighted mean relative *active* power branch flow error as defined in equation (2) in [1].

error_flow_s: (hynet_float_) Contribution-weighted mean relative *apparent* power branch flow error.

Return type pandas.Series

References

hynet.reduction.large_scale.evaluation.**show_reduction_evaluation**(*evaluation*)

Show the evaluation of a series of reduction stages or a parameter sweep.

Parameters **evaluation** (*pandas.DataFrame*) – Data frame with information on the extent and accuracy of the individual reduction stages or parameters. This data frame must be indexed by the stage names / parameter values and comprise the following columns:

bus_reduction: (hynet_float_) Ratio of the number of reduced buses w.r.t. the total number of buses before the reduction.

branch_reduction: (hynet_float_) Ratio of the number of reduced branches w.r.t. the total number of branches before the reduction.

cycle_reduction: (hynet_float_) Ratio of the number of reduced cycles w.r.t. the total number of cycles before the reduction.

error_disp: (hynet_float_) Contribution-weighted mean relative *active* power dispatch error as defined in equation (1) in [1].

error_flow: (hynet_float_) Contribution-weighted mean relative *active* power branch flow error as defined in equation (2) in [1].

Returns fig

Return type matplotlib.figure.Figure

References

hynet.reduction.large_scale.features module

Declaration of features for the network reduction.

hynet.reduction.large_scale.features.**add_branch_features**(*scenario*, *length_threshold*)

Add the standard branch-related features.

The transformers as well as long branches are marked as features.

Parameters

- **scenario** (Scenario) – Scenario that shall be processed.
- **length_threshold** (*float*) – Threshold in kilometers on the length of a branch to consider it as long.

hynet.reduction.large_scale.features.**add_bus_features**(*scenario*)

Add the standard bus-related features.

The terminal buses of conventional generators as well as all reference buses are marked as features.

Parameters **scenario** (Scenario) – Scenario that shall be processed.

hynet.reduction.large_scale.features.**add_congestion_features**(*scenario*, *result*, *loading_threshold*, *dv_threshold*)

Add the standard congestion-related features.

Branches that are highly loaded or that exhibit binding constraints are marked as features.

Parameters

- **scenario** (Scenario) – Scenario that shall be processed.
- **result** (OPFResult) – OPF result of the scenario.
- **loading_threshold** (*float*) – Relative threshold on the branch flow w.r.t. its effective rating to consider it as highly loaded.
- **dv_threshold** (*float*) – Threshold on the dual variables of the ampacity and angle difference constraint.

hynet.reduction.large_scale.features.**add_converter_features**(*scenario*)

Add the standard converter-related features.

The terminal buses of converters are marked as features.

Parameters **scenario** (Scenario) – Scenario that shall be processed.

hynet.reduction.large_scale.features.**add_feature_columns**(*scenario*)

Ensure that the provided scenario contains the feature columns.

For the network reduction, buses and branches of particular interest can be marked as features of the system to prevent their reduction. This function adds the respective columns to the data frames of the scenario, in case that they do no exist already.

Parameters **scenario** (Scenario) – Scenario that should contain the feature columns.

Returns False if one or more feature columns already existed (i.e., feature information is present) and True otherwise.

Return type bool

hynet.reduction.large_scale.features.**add_standard_features**(*scenario*, *result*, *length_threshold=50*, *loading_threshold=0.8*, *dv_threshold=1.0*)

Add the standard features for feature-preserving network reduction.

The feature-preserving network reduction permits the preservation of certain entities of particular importance. This function marks the entities described in Section III in [1] as features, i.e., it adds the standard bus-, branch-, converter- and injector-related features to the scenario.

Parameters

- **scenario** (Scenario) – Scenario that shall be processed.
- **result** (OPFResult) – OPF result of the scenario.
- **length_threshold** (*float, optional*) – Threshold in kilometers on the length of a branch to consider it as long (default is 50km).
- **loading_threshold** (*float, optional*) – Relative threshold on the branch flow w.r.t. its effective rating to consider it as highly loaded (default is 0.8, i.e., 80% utilization).
- **dv_threshold** (*float, optional*) – Threshold on the dual variables of the ampacity and angle difference constraint (default is 1.0).

See also:

hynet.reduction.large_scale.features.add_feature_columns() Add the feature columns to the scenario.

hynet.reduction.large_scale.features.add_bus_features() Add the standard bus-related features.

hynet.reduction.large_scale.features.add_converter_features() Add the standard converter-related features.

hynet.reduction.large_scale.features.add_branch_features() Add the standard branch-related features.

hynet.reduction.large_scale.features.add_congestion_features() Add the standard congestion-related features.

References

hynet.reduction.large_scale.features.**count_features**(*scenario*)

Return the number of features of the scenario.

Parameters **scenario** (Scenario) – Scenario that shall be processed.

Returns The number of bus and branch features.

Return type int

hynet.reduction.large_scale.features.**has_branch_features**(*scenario*, *subgrid*)

Return True if the subgrid contains any branch features.

Parameters

- **scenario** (Scenario) – Scenario that contains the subgrid.

- **subgrid** (*list[hynet_id_]*) – List with the bus IDs of the subgrid.

Returns `True` if the subgrid contains any branch features and `False` otherwise.

Return type bool

hynet.reduction.large_scale.features.**has_bus_features**(*scenario*, *subgrid*)
Return `True` if the subgrid contains any bus features.

Parameters

- **scenario** (Scenario) – Scenario that contains the subgrid.
- **subgrid** (*list[hynet_id_]*) – List with the bus IDs of the subgrid.

Returns `True` if the subgrid contains any bus features and `False` otherwise.

Return type bool

hynet.reduction.large_scale.features.**has_features**(*scenario*, *subgrid*)
Return `True` if the subgrid contains any bus or branch features.

Parameters

- **scenario** (Scenario) – Scenario that contains the subgrid.
- **subgrid** (*list[hynet_id_]*) – List with the bus IDs of the subgrid.

Returns `True` if the subgrid contains any bus or branch features and `False` otherwise.

Return type bool

hynet.reduction.large_scale.market module

Market-based subgrid selection and reduction.

hynet.reduction.large_scale.market.**combine_overlapping_clusters**(*candidates*)
Combine any overlapping clusters to a single cluster.

hynet.reduction.large_scale.market.**identify_price_clusters**(*opf_result*, *max_price_diff*)
Identify clusters of buses that exhibit a similar nodal price.

The dual variable of the active power balance constraint is considered as the nodal price (i.e., the locational marginal price if the OPF problem exhibits strong duality). After the price clusters were identified, overlapping clusters are combined, such that a bus of the system may only be part of one cluster.

Parameters

- **opf_result** (OPFResult) – OPF result based on which price clusters shall be identified.
- **max_price_diff** (*float*) – Threshold δ in \$/MW on fluctuations of the nodal price (LMP) from the center of a price cluster to consider the respective buses to be part of the cluster.

Returns

clusters – Data frame with information on the price clusters, whose index consists of the bus ID of the cluster centers and the following columns:

cluster: (hynet_int_) Number of cycles in the subgrid.

price_ref: (.hynet_float_) Reference price for the cluster. Note that, due to the combining of overlapping clusters, the price difference within a cluster may exceed the threshold on the price fluctuations.

Return type pandas.DataFrame

hynet.reduction.large_scale.market.**reduce_by_market**(*scenario*, *opf_result*, *max_price_diff*)

Apply a market-based network reduction to the scenario.

This function performs the marked-based network reduction described in Section IV-D in [1].

Parameters

- **scenario** (Scenario) – Scenario that shall be processed.
- **opf_result** (OPFResult) – OPF result of the provided scenario.
- **max_price_diff** (*float*) – Threshold δ in \$/MW on fluctuations of the nodal price (LMP) from the center of a price cluster to consider the respective buses to be part of the cluster (cf. equation (4) in [1]).

Returns Number of buses that were reduced.

Return type int

References

hynet.reduction.large_scale.subgrid module

Subgrid reduction: Aggregate the buses of a subgrid to the representative bus.

hynet.reduction.large_scale.subgrid.**create_bus_id_map**(*scenario*)

Create a mapping from the original buses to the reduced buses.

During the network reduction, certain buses are aggregated at other buses. This function creates a mapping from the buses of the original system to the buses of the reduced system.

Parameters **scenario** (Scenario) – Scenario that was subject to network reduction.

Returns **bus_id_map** – This series is *indexed* by the bus IDs *before* network reduction, while the *values* are the bus IDs *after* network reduction, i.e., it maps the buses of the original system to the buses in the reduced system.

Return type pandas.Series

hynet.reduction.large_scale.subgrid.**preserve_aggregation_info**(*scenario*)

Copy the information on aggregated buses to the bus annotation.

During the network reduction, certain buses are aggregated at other buses, where the aggregated buses are documented in the column `aggregation`. As this is an extension to *hynet*'s scenario format, it is not stored to the grid database upon saving. To this end, this information is preserved using the bus annotation.

Parameters **scenario** (Scenario) – Scenario that was subject to network reduction.

hynet.reduction.large_scale.subgrid.**reduce_subgrid**(*scenario*, *representative_bus*, *subgrid*)

Reduce the subgrid buses to the representative bus, if it contains no features.

If the subgrid *does not contain any features*, it is reduced to the representative bus as described in Section IV-A in [1]. In case of the latter, a column `aggregation` is added/updated in the bus data frame that contains a list with the bus ID of all buses that were aggregated to the respective bus.

Parameters

- **scenario** (Scenario) – Scenario that shall be processed.
- **representative_bus** (*hynet_id_*) – Bus ID of the representative bus.
- **subgrid** (*Iterable[hynet_id_]*) – Iterable with the bus IDs of the subgrid that shall be aggregated at the representative bus. It *must not* contain the representative bus.

References

hynet.reduction.large_scale.subgrid.**restore_aggregation_info**(*scenario*)
Restore the information on aggregated buses from the bus annotation.

Parameters **scenario** (Scenario) – Scenario that was subject to network reduction.

See also:

hynet.reduction.large_scale.subgrid.preserve_aggregation_info()

hynet.reduction.large_scale.sweep module

Parameter sweeps to assist the selection of an appropriate network reduction.

hynet.reduction.large_scale.sweep.**sweep_feature_depth**(*scenario*, *rel_impedance_thres*, *values=None*, *critical_mw_diff=None*, *server=None*, *solver=None*, *opf_reference=None*, *show_evaluation=True*)
Sweep the feature depth ϑ

After and initial electrical coupling-based network reduction, additional bus features may be added to reduce the dispatch error at critical injectors, which is parameterized by the feature depth ϑ, cf. Section V-A and Fig. 5 in [1]. To assist the selection of an appropriate depth, this function provides the evaluation of the two-stage electrical coupling-based network reduction for different feature depths.

If the `feature` columns are not present upon the call of this function, the standard features described in Section III in [1] are considered.

Parameters

- **scenario** (Scenario) – Scenario that shall be evaluated.
- **rel_impedance_thres** (*float, optional*) – Relative threshold τ for the electrical coupling-based network reduction.
- **values** (*list[int], optional*) – List of (ascending) values for the feature depth ϑ. By default, the depths 0 to 10 are considered.
- **critical_mw_diff** (*float, optional*) – Absolute threshold in MW on the active power dispatch difference of an injector to consider it as critical. The default is 0.02% of the total active power load.
- **server** (OptimizationServer, *optional*) – *hynet* optimization server for the computation of the OPFs. By default, a server in local mode is used.
- **solver** (SolverInterface, *optional*) – Solver for the OPF problems. The default selects the most appropriate QCQP solver among those installed.
- **opf_reference** (OPFResult, *optional*) – OPF result of the scenario *before* the reduction to evaluate the reduction accuracy. By default, an OPF for the provided scenario is utilized.
- **show_evaluation** (*bool, optional*) – If `True` (default), the results of the sweep are visualized.

Returns

evaluation – Data frame with information on the extent and accuracy of the reduction for the individual feature depths. This data frame is indexed by the depth and comprises the following columns:

bus_reduction: (**hynet_float_**) Ratio of the number of reduced buses w.r.t. the total number of buses before the reduction.

branch_reduction: (**hynet_float_**) Ratio of the number of reduced branches w.r.t. the total number of branches before the reduction.

cycle_reduction: (**hynet_float_**) Ratio of the number of reduced cycles w.r.t. the total number of cycles before the reduction.

error_disp: (**hynet_float_**) Contribution-weighted mean relative *active* power dispatch error as defined in equation (1) in [1].

error_disp_s: (**hynet_float_**) Contribution-weighted mean relative *apparent* power dispatch error.

error_flow: (**hynet_float_**) Contribution-weighted mean relative *active* power branch flow error as defined in equation (2) in [1].

error_flow_s: (**hynet_float_**) Contribution-weighted mean relative *apparent* power branch flow error.

opf: (**hynet.OPFResult**) OPF result for the respective scenario.

Return type pandas.DataFrame

References

hynet.reduction.large_scale.sweep.**sweep_max_price_diff**(*scenario*, *values*, *server=None*, *solver=None*, *opf_reference=None*, *show_evaluation=True*)

Sweep the maximum nodal price fluctuation δ for clustering.

The market-based network reduction is parameterized by the threshold δ in \$/MW on fluctuations of the nodal price (LMP) from the center of a price cluster, cf. Section IV-D and (4) in [1]. To assist the selection of an appropriate threshold, this function provides the evaluation of the market-based network reduction for different threshold values (cf. Fig. 6 in [1]).

If the `feature` columns are not present upon the call of this function, the standard features described in Section III in [1] are considered.

Parameters

- **scenario** (`Scenario`) – Scenario that shall be evaluated.
- **values** (*`list[float]`*) – List of (ascending) values for the threshold δ in \$/MW on fluctuations of the nodal price. To identify a reasonable range for the threshold values, it may be helpful to calculate an OPF and inspect the LMP profile via `hynet.show_lmp_profile`.
- **server** (`OptimizationServer`, *`optional`*) – *hynet* optimization server for the computation of the OPFs. By default, a server in local mode is used.
- **solver** (`SolverInterface`, *`optional`*) – Solver for the OPF problems. The default selects the most appropriate QCQP solver among those installed.
- **opf_reference** (`OPFResult`, *`optional`*) – OPF result of the scenario *before* the reduction to evaluate the reduction accuracy. By default, an OPF for the provided scenario is utilized.
- **show_evaluation** (*`bool`*, *`optional`*) – If `True` (default), the results of the sweep are visualized.

Returns

evaluation – Data frame with information on the extent and accuracy of the reduction for the individual threshold values. This data frame is indexed by the threshold and comprises the following columns:

bus_reduction: (**hynet_float_**) Ratio of the number of reduced buses w.r.t. the total number of buses before the reduction.

branch_reduction: (**hynet_float_**) Ratio of the number of reduced branches w.r.t. the total number of branches before the reduction.

cycle_reduction: (**hynet_float_**) Ratio of the number of reduced cycles w.r.t. the total number of cycles before the reduction.

error_disp: (**hynet_float_**) Contribution-weighted mean relative *active* power dispatch error as defined in equation (1) in [1].

error_disp_s: (**hynet_float_**) Contribution-weighted mean relative *apparent* power dispatch error.

error_flow: (**hynet_float_**) Contribution-weighted mean relative *active* power branch flow error as defined in equation (2) in [1].

error_flow_s: (**hynet_float_**) Contribution-weighted mean relative *apparent* power branch flow error.

opf: (**hynet.OPFResult**) OPF result for the respective scenario.

Return type pandas.DataFrame

References

hynet.reduction.large_scale.sweep.**sweep_rel_impedance_thres**(*scenario*, *values=None*, *server=None*, *solver=None*, *opf_reference=None*, *show_evaluation=True*)

Sweep the relative impedance threshold :math:‘ au‘.

The electrical coupling-based network reduction is parameterized by the relative impedance threshold :math:‘ au‘, cf. Section IV-C and (3) in [1]. To assist the selection of an appropriate threshold, this function provides the evaluation of the electrical coupling-based network reduction for different threshold values (cf. Fig. 4 in [1]).

If the `feature` columns are not present upon the call of this function, the standard features described in Section III in [1] are considered.

Parameters

- **scenario** (*hynet.Scenario*) – Scenario that shall be evaluated.
- **values** (*list[float], optional*) – List of (ascending) values for the relative impedance threshold τ. By default, the 15 values in the interval [0.005, 0.1] shown in Fig. 4 in [1] are considered.
- **server** (*OptimizationServer, optional*) – *hynet* optimization server for the computation of the OPFs. By default, a server in local mode is used.
- **solver** (*SolverInterface, optional*) – Solver for the OPF problems. The default selects the most appropriate QCQP solver among those installed.
- **opf_reference** (*OPFResult, optional*) – OPF result of the scenario *before* the reduction to evaluate the reduction accuracy. By default, an OPF for the provided scenario is utilized.
- **show_evaluation** (*bool, optional*) – If `True` (default), the results of the sweep are visualized.

Returns

evaluation – Data frame with information on the extent and accuracy of the reduction for the individual threshold values. This data frame is indexed by the threshold and comprises the following columns:

bus_reduction: (**hynet_float_**) Ratio of the number of reduced buses w.r.t. the total number of buses before the reduction.

branch_reduction: (**hynet_float_**) Ratio of the number of reduced branches w.r.t. the total number of branches before the reduction.

cycle_reduction: (**hynet_float_**) Ratio of the number of reduced cycles w.r.t. the total number of cycles before the reduction.

error_disp: (**hynet_float_**) Contribution-weighted mean relative *active* power dispatch error as defined in equation (1) in [1].

error_disp_s: (**hynet_float_**) Contribution-weighted mean relative *apparent* power dispatch error.

error_flow: (**hynet_float_**) Contribution-weighted mean relative *active* power branch flow error as defined in equation (2) in [1].

error_flow_s: (**hynet_float_**) Contribution-weighted mean relative *apparent* power branch flow error.

opf: (**hynet.OPFResult**) OPF result for the respective scenario.

Return type pandas.DataFrame

References

hynet.reduction.large_scale.topology module

Topology-based subgrid selection and reduction.

hynet.reduction.large_scale.topology.**identify_islands**(*scenario*, *max_island_size*)

Identify "islands" at the boundary of the grid.

In the context of the topology-based network reduction, "islands" are clusters of buses that are connected to the main grid via a single corridor. This function identifies these "islands" by testing every corridor, i.e., the branches in the corridor are removed, islanding of a subgrid is detected and, if its size does not exceed the defined maximum island size, it is considered an "island".

Parameters

- **scenario** (Scenario) – Scenario that shall be examined.
- **max_island_size** (*int*) – Maximum size of an "island" in terms of its number of buses.

Returns **island_info** – Information on the "islands" within the system. The returned list contains an item for every island, which is a tuple comprising the island's representative bus (i.e., the bus of the main grid to which the island is connected) and a pandas index with the bus ID of all buses of the island.

Return type list[tuple(hynet_id_, pandas.Index[hynet_id_])]

hynet.reduction.large_scale.topology.**reduce_by_topology**(*scenario*, *max_island_size=None*)

Apply a topology-based network reduction to the scenario.

This function performs the topology-based network reduction described in Section IV-B in [1].

Parameters

- **scenario** (Scenario) – Scenario that shall be processed.
- **max_island_size** (*int, optional*) – Maximum size of an "island" in terms of its number of buses. By default, it is set to approximately 1% of the total number of buses. Set this parameter to `0` to disable the reduction of "islands".

Returns Number of buses that were reduced.

Return type int

References

hynet.reduction.large_scale.topology.**reduce_islands**(*scenario*, *max_island_size*)
Reduce "islands" within the scenario.

This function reduces "islands" at the boundary of the grid, see Section IV-B in [1].

Parameters

- **scenario** (Scenario) – Scenario that shall be processed.
- **max_island_size** (*int*) – Maximum size of an "island" in terms of its number of buses.

Returns Number of buses that were reduced.

Return type int

References

hynet.reduction.large_scale.topology.**reduce_single_buses**(*scenario*)
Reduce single buses within the scenario.

This function reduces buses that only exhibit one adjacent bus, see Section IV-B in [1]. This function may be called repeatedly to reduce "lines of buses" as considered in [1].

Parameters **scenario** (Scenario) – Scenario that shall be processed.

Returns Number of buses that were reduced.

Return type int

References

hynet.reduction.large_scale.utilities module

Utility functions for the network reduction.

hynet.reduction.large_scale.utilities.**add_adjacent_bus_info**(*scenario*)
Add information about adjacent buses to the scenario.

This function adds/updates the following columns to the bus data frame:

num_adjacent: (int) Number of adjacent buses.

adjacent: (pandas.Index[hynet_id_]) Pandas index with the bus ID of all adjacent buses.

Parameters **scenario** (Scenario) – Scenario that shall be processed.

hynet.reduction.large_scale.utilities.**add_parallel_branch_info**(*scenario*)
Add information about parallel branches to the scenario.

This function adds/updates the following columns to the branch data frame:

parallel: (bool) `True` if the branch is parallel to another one and `False` otherwise.

parallel_main_id: (hynet_id_) Branch ID of the (arbitrarily selected) main branch among the group of parallel branches. With this information, groups of parallel branches can be determined by filtering this column w.r.t. the main branch ID.

Parameters **scenario** (Scenario) – Scenario that shall be processed.

hynet.reduction.large_scale.utilities.**remove_adjacent_bus_info**(*scenario*)
Remove the information about adjacent buses.

Parameters **scenario** (Scenario) – Scenario that shall be processed.

hynet.reduction.large_scale.utilities.**remove_parallel_branch_info**(*scenario*)
Remove the information about parallel branches.

Parameters **scenario** (Scenario) – Scenario that shall be processed.

Module contents

Feature- and structure-preserving network reduction for large-scale grids.

This subpackage of *hynet* implements the reduction strategy for large-scale grid models as introduced in [1], which aims at reducing model complexity while preserving relevant features and the relation to the structure of the original model. Please refer to [1] for more details.

References

3.1.7.2 Submodules

3.1.7.3 hynet.reduction.copper_plate module

Network reduction to a "copper plate".

hynet.reduction.copper_plate.**reduce_to_copper_plate**(*scenario*)
Return a deep copy of the scenario with the grid reduced to a copper plate.

In the "copper plate" reduction, every connected grid is reduced to a single bus, i.e., the impact of the power grid is neglected and the network laws are reduced to simply balance the total injection and total load.

3.1.7.4 Module contents

Network reduction methods in *hynet*.

3.1.8 hynet.scenario package

3.1.8.1 Submodules

3.1.8.2 hynet.scenario.capability module

Representation of a capability region of injectors and converters.

class hynet.scenario.capability.**CapRegion**(*p_bnd=None*, *q_bnd=None*, *lt=None*, *rt=None*, *lb=None*, *rb=None*)
Bases: object

Specification of a capability region.

The capability region is the intersection of a box and a polyhedron. Let `s = [p, q]^T` be the vector of active and reactive power injection. The capability region `S` is the intersection of `S_box` and `S_phs`, where

```
(1) S_box = { s in R^2: p_min <= p <= p_max, q_min <= q <= q_max }
(2) S_phd = { s in R^2: A*s <= b }.
```

The polyhedron `S_phd` is defined by the properties `lt`, `lb`, `rt`, and `rb`, which are half-spaces that are defined by a *nonnegative* relative offset and a slope, see the parameter description below. The parametrization in `(2)` can be obtained using the method `get_polyhedron`.

Parameters

- **p_min** (*hynet_float_*) – Active power lower bound in MW.
- **p_max** (*hynet_float_*) – Active power upper bound in MW.
- **q_min** (*hynet_float_*) – Reactive power lower bound in Mvar.
- **q_max** (*hynet_float_*) – Reactive power upper bound in Mvar.
- **lt** (*HalfSpace or None*) – Left-top half-space or `None` if omitted. This half-space is anchored at `(p,q)` with `p = p_min` and `q = lt.offset * q_max`. The slope must be *positive*.
- **rt** (*HalfSpace or None*) – Right-top half-space or `None` if omitted. This half-space is anchored at `(p,q)` with `p = p_max` and `q = rt.offset * q_max`. The slope must be *negative*.
- **lb** (*HalfSpace or None*) – Left-bottom half-space or `None` if omitted. This half-space is anchored at `(p,q)` with `p = p_min` and `q = lb.offset * q_min`. The slope must be *negative*.
- **rb** (*HalfSpace or None*) – Right-bottom half-space or `None` if omitted. This half-space is anchored at `(p,q)` with `p = p_max` and `q = rb.offset * q_min`. The slope must be *positive*.

add_power_factor_limit(*power_factor*)
Add a left-top and left-bottom half-space to limit the power factor.

Parameters **power_factor** (*hynet_float_*) – Power factor limit.

copy()
Return a deep copy of this capability region.

edit()
Edit this capability region in the capability region visualizer.

Caution: Due to technical reasons, all open `matplotlib` figures are closed during this call.

Remark: In case you are using MAC OS X, please be aware of this issue with `matplotlib` and `tkinter`, which causes Python to crash if the capability region visualizer is opened. To avoid it, set `matplotlib`'s backend to `TkAgg` *before* importing *hynet* using

```
>>> import matplotlib
>>> matplotlib.use('TkAgg')
```

get_polyhedron()
Returns the polyhedron formulation `S_phd = { s in R^2: A*s <= b }`.

Returns

- **A** (*np.ndarray*) – `A` in the formulation of the polyhedron above.
- **b** (*np.ndarray*) – `b` in the formulation of the polyhedron above.
- **name** (*list[str]*) – Abbreviation of the corresponding half-space for each row.

has_polyhedron()
Return `True` if any of the half-spaces is set.

lb
Return the left-bottom half-space.

lt
Return the left-top half-space.

rb
Return the right-bottom half-space.

rt
Return the right-top half-space.

scale(*scaling*)
Scale the upper and lower bound on active and reactive power.

show(*operating_point=None*)
Show this capability region in the capability region visualizer.

Caution: Due to technical reasons, all open `matplotlib` figures are closed during this call.

Remark: In case you are using MAC OS X, please be aware of this issue with `matplotlib` and `tkinter`, which causes Python to crash if the capability region visualizer is opened. To avoid it, set `matplotlib`'s backend to `TkAgg` *before* importing *hynet* using

```
>>> import matplotlib
>>> matplotlib.use('TkAgg')
```

Parameters **operating_point** (*hynet_complex_, optional*) – If provided, a marker is shown for this operating point in the P/Q-plane.

class hynet.scenario.capability.**ConverterCapRegion**(*p_bnd=None*, *q_bnd=None*, *lt=None*, *rt=None*, *lb=None*, *rb=None*)

Bases: *hynet.scenario.capability.CapRegion*

Specification of a converter capability region.

The capability region is specified in terms of the apparent power flow at the respective terminal bus of the converter, see `CapRegion` for more information. With respect to the parameters returned by `get_polyhedron` and `get_box_constraint`, this class considers the converter state variable `f = [p_fwd, p_bwd, q_src, q_dst]^T` instead of the apparent power vector `s = [p, q]^T` considered in `CapRegion`.

See also:

hynet.scenario.capability.CapRegion

add_power_factor_limit(*power_factor*)
Add a left-top and left-bottom half-space to limit the power factor.

Parameters **power_factor** (*hynet_float_*) – Power factor limit.

static get_box_constraint(*cap_src*, *cap_dst*, *loss_factor_fwd*, *loss_factor_bwd*)
Return `(f_lb, f_ub)` of the state constraint `f_lb <= f <= f_ub`.

A converter comprises two capability regions, one at the source terminal and one at the destination terminal. Thus, the box constraint on the converter state variable is a combination of both as well as the conversion loss factors.

Parameters

- **cap_src** (ConverterCapRegion) – Source terminal capability region.
- **cap_dst** (ConverterCapRegion) – Destination terminal capability region.
- **loss_factor_fwd** (*float*) – Proportional loss factor for the *forward* flow.
- **loss_factor_bwd** (*float*) – Proportional loss factor for the *backward* flow.

Returns

- **f_lb** (*np.ndarray[.hynet_float_]*) – Lower bound on the converter state vector.

- **f_ub** (*np.ndarray[.hynet_float_]*) – Upper bound on the converter state vector.

get_polyhedron(*terminal*, *loss_factor*)
Returns the polyhedron formulation `S_phd = { f in R^4:  A*f <= b }`.

In addition to the half-spaces of the capability region, this polyhedron includes, for bidirectional converters with a nonnegligible capacity, an additional constraint to limit the noncomplementary operation of the converter.

Parameters

- **terminal** (*str*) – Terminal of the converter (`'src'` or `'dst'`) with which the capability region is associated.
- **loss_factor** (*float*) – Proportional loss factor for (a) the *backward* flow if the terminal is `'src'` and (b) the *forward* flow if the terminal is `'dst'`.

Returns

- **A** (*np.ndarray*) – `A` in the formulation of the polyhedron above.
- **b** (*np.ndarray*) – `b` in the formulation of the polyhedron above.
- **name** (*list[str]*) – Abbreviation of the corresponding half-space or active power limit for each row.

See also:

hynet.scenario.capability.ConverterCapRegion.get_box_constraint()

class hynet.scenario.capability.**HalfSpace**
Bases: *hynet.scenario.capability.HalfSpace*

Representation of a half-space in a capability region.

The half-spaces of a capability region are specified in terms of a nonnegative relative offset "`offset`" w.r.t. to the corresponding reactive power limit (i.e., the absolute offset is the reactive power limit multiplied by `offset`) and its slope "`slope`". In the dimension of active power, it is anchored at the corresponding active power limit.

3.1.8.3 hynet.scenario.cost module

Representation of (injector) cost functions in *hynet*.

class hynet.scenario.cost.**PWLFunction**(*samples=None*, *marginal_price=None*)
Bases: `object`

Representation of a piecewise linear function `f:  R -> R`.

Parameters **samples** (*(numpy.ndarray[hynet_float_], numpy.ndarray[hynet_float_]), optional*) – Tuple `(x, y)` of `x`- and `y`-coordinates of sample points of the piecewise linear function, i.e., `(x[0], y[0]), ..., (x[N], y[N])`.

evaluate(*x*)
Evaluate the function at `x`, i.e., return `y = f(x)`.

get_epigraph_polyhedron()
Return `(A, b)` such that `f(x) = min{z in R: z*1 >= A*x + b}`.

Note that `f` must be **convex**.

is_convex()
Return `True` if the function is convex.

marginal_price
Return the slope if the function is linear or `numpy.nan` otherwise.

Remark: This function requires that the linear function is specified using two sample points, i.e., the function will return `numpy.nan` even if three or more samples lie on a line. The latter case should be avoided anyway, as it introduces unnecessary constraints in the epigraph representation of the function.

samples
Return the tuple `(x, y)` of `x`- and `y`-coordinate arrays.

scale(*scaling*)
Scale the function by `scaling`, i.e., `f(x) -> scaling * f(x)`.

show()
Show the piecewise linear function.

3.1.8.4 hynet.scenario.representation module

Representation of a steady-state scenario in *hynet*.

class hynet.scenario.representation.**Scenario**
Bases: `object`

Specification of a steady-state grid scenario.

Parameters

- **id** (*hynet_id_*) – Identifier of the scenario.
- **name** (*str*) – Name of the scenario.
- **time** (*hynet_float_*) – Time in hours, relative to the scenario collection start time.
- **database_uri** (*str*) – URI of the associated *hynet* grid database.
- **grid_name** (*str*) – Name of the grid.
- **base_mva** (*hynet_float_*) – Apparent power normalization constant in MVA.
- **loss_price** (*hynet_float_*) – Artificial price for losses in $/MWh. The corresponding cost of losses is taken into account for the minimum-cost dispatch.
- **description** (*str*) – Description of the grid database.
- **annotation** (*str*) – Annotation string for supplementary information.
- **bus** (*pandas.DataFrame*) – Data frame with one data set per bus, indexed by the *bus ID*, which comprises the following columns:

  type: (BusType) Type of voltage waveform at the bus.

  ref: (bool) `True` if the bus serves as a reference.

  base_kv: (hynet_float_) Base voltage in kV.

  y_tld: (hynet_complex_) Shunt admittance in p.u.. For example, to add a shunt compensator that injects `q` Mvar and dissipates `p` MW at 1 p.u., set `y_tld` to `(p + 1j*q) / base_mva`.

  load: (hynet_complex_) Aggregated inelastic apparent power load in MVA.

  v_min: (hynet_float_) Voltage lower bound in p.u..

  v_max: (hynet_float_) Voltage upper bound in p.u..

  zone: (hynet_id_ or None) Zone ID or `None` if not available.

  annotation: (str) Annotation string for supplementary information.

- **branch** (*pandas.DataFrame*) – Data frame with one data set per branch, indexed by the *branch ID*, which comprises the following columns:

  type: (BranchType) Type of entity modeled by the branch.

src: (**hynet_id_**) Source bus ID.

dst: (**hynet_id_**) Destination bus ID.

z_bar: (**hynet_complex_**) Series impedance of the pi-equivalent in p.u..

y_src: (**hynet_complex_**) Shunt admittance at the source side of the pi-equivalent in p.u..

y_dst: (**hynet_complex_**) Shunt admittance at the destination side of the pi-equivalent in p.u..

rho_src: (**hynet_complex_**) Complex voltage ratio of the ideal transformer at the source side.

rho_dst: (**hynet_complex_**) Complex voltage ratio of the ideal transformer at the destination side.

length: (**hynet_float_**) Line length in kilometers or `numpy.nan` if not available.

rating: (**hynet_float_**) Ampacity in terms of a long-term MVA rating at a bus voltage of 1 p.u. (i.e., the current limit is `rating/base_mva`) or `numpy.nan` if omitted.

angle_min: (**hynet_float_**) Angle difference lower bound in degrees or `numpy.nan` if omitted.

angle_max: (**hynet_float_**) Angle difference upper bound in degrees or `numpy.nan` if omitted.

drop_min: (**hynet_float_**) Voltage drop lower bound in percent or `numpy.nan` if omitted.

drop_max: (**hynet_float_**) Voltage drop upper bound in percent or `numpy.nan` if omitted.

annotation: (**str**) Annotation string for supplementary information.

- **converter** (*pandas.DataFrame*) – Data frame with one data set per converter, indexed by the *converter ID*, which comprises the following columns:

  src: (**hynet_id_**) Source bus ID.

  dst: (**hynet_id_**) Destination bus ID.

  cap_src: (**ConverterCapRegion**) Specification of the converter's capability region in the P/Q-plane (in MW and Mvar, respectively) at the source bus. The P-axis is the active power flow *into the converter* and the Q-axis is the reactive power injection *into the grid*.

  cap_dst: (**ConverterCapRegion**) Specification of the converter's capability region in the P/Q-plane (in MW and Mvar, respectively) at the destination bus. The P-axis is the active power flow *into the converter* and the Q-axis is the reactive power injection *into the grid*.

  loss_fwd: (**hynet_float_**) Loss factor in percent for the *forward flow* of active power. This proportional loss factor describes the dynamic conversion losses if active power flows from the source bus to the destination bus.

  loss_bwd: (**hynet_float_**) Loss factor in percent for the *backward flow* of active power. This proportional loss factor describes the dynamic conversion losses if active power flows from the destination bus to the source bus.

  loss_fix: (**hynet_float_**) Static losses in MW (considered at the converter's source bus).

  annotation: (**str**) Annotation string for supplementary information.

- **injector** (*pandas.DataFrame*) – Data frame with one data set per injector, indexed by the *injector ID*, which comprises the following columns:

  type: (InjectorType) Type of entity modeled by the injector.

  bus: (hynet_id_) Terminal bus ID.

  cap: (CapRegion) Specification of the injector's capability region in the P/Q-plane (in MW and Mvar, respectively).

  cost_p: (PWLFunction or None) Piecewise linear cost function for active power, which specifies the cost in dollars for an active power injection in MW, or `None` in case of zero costs.

  cost_q: (PWLFunction or None) Piecewise linear cost function for reactive power, which specifies the cost in dollars for a reactive power injection in Mvar, or `None` in case of zero costs.

  cost_start: (hynet_float_) Startup cost in dollars.

  cost_stop: (hynet_float_) Shutdown cost in dollars.

  ramp_up: (hynet_float_) Maximum upramping rate for active power in MW/h or `numpy.nan` if the upramping rate is not limited.

  ramp_down: (hynet_float_) Maximum downramping rate for active power in MW/h or `numpy.nan` if the downramping rate is not limited.

  min_up: (hynet_float_) Minimum uptime in hours or `numpy.nan` if it is not limited.

  min_down: (hynet_float_) Minimum downtime in hours or `numpy.nan` if it is not limited.

  energy_min: (hynet_float_) Energy lower bound in MWh or `numpy.nan` if it is not limited.

  energy_max: (hynet_float_) Energy upper bound in MWh or `numpy.nan` if it is not limited.

  annotation: (str) Annotation string for supplementary information.

See also:

hynet.data.interface.load_scenario Load a scenario from a *hynet* grid database.

add_branch(*type_*, *src*, *dst*, *z_bar*, *y_src=0.0*, *y_dst=0.0*, *rho_src=1.0*, *rho_dst=1.0*, *length=nan*, *rating=nan*, *angle_min=nan*, *angle_max=nan*, *drop_min=nan*, *drop_max=nan*, *annotation=''*, *branch_id=None*)

Add a branch to this scenario.

Please refer to the documentation of the `branch` data frame for a description of the parameters. If `branch_id` is provided, this ID is used for the branch, otherwise an appropriate ID is generated.

Returns **branch_id** – ID of the added branch.

Return type hynet_id_

add_bus(*type_*, *base_kv*, *v_min*, *v_max*, *ref=False*, *y_tld=0*, *load=0*, *zone=None*, *annotation=''*, *bus_id=None*)

Add a bus to this scenario.

Please refer to the documentation of the `bus` data frame for a description of the parameters. If `bus_id` is provided, this ID is used for the bus, otherwise an appropriate ID is generated.

Returns **bus_id** – ID of the added bus.

Return type hynet_id_

add_compensator(*bus*, *q_max*, *q_min=None*, *cost_q=None*)
Add a reactive power compensator to this scenario.

Parameters

- **bus** (*hynet_id_*) – Bus ID of the terminal bus for the compensator.
- **q_max** (*hynet_float_*) – Maximum reactive power in Mvar that can be injected.
- **q_min** (*hynet_float_, optional*) – Lower bound in Mvar (default `-q_max`) on the reactive power injection.
- **cost_q** (PWLFunction, *optional*) – Piecewise linear cost function for reactive power. By default, the reactive power is provided at zero cost.

Returns **injector_id** – Injector ID of the added compensator.

Return type hynet_id_

add_converter(*src*, *dst*, *cap_src*, *cap_dst*, *loss_fwd=0*, *loss_bwd=0*, *loss_fix=0*, *annotation=''*, *converter_id=None*)
Add a converter to this scenario.

Please refer to the documentation of the `converter` data frame for a description of the parameters. If `converter_id` is provided, this ID is used for the converter, otherwise an appropriate ID is generated.

Returns **converter_id** – ID of the added converter.

Return type hynet_id_

add_injector(*type_*, *bus*, *cap*, *cost_p=None*, *cost_q=None*, *cost_start=0*, *cost_stop=0*, *ramp_up=nan*, *ramp_down=nan*, *min_up=nan*, *min_down=nan*, *energy_min=nan*, *energy_max=nan*, *annotation=''*, *injector_id=None*)
Add an injector to this scenario.

Please refer to the documentation of the `injector` data frame for a description of the parameters. If `injector_id` is provided, this ID is used for the injector, otherwise an appropriate ID is generated.

Returns **injector_id** – ID of the added injector.

Return type hynet_id_

analyze_cycles()
Return a data frame with an analysis of cyclic connections of buses.

The information about cyclic connections of buses is relevant in the study of transitions to the *hybrid architecture*, which establishes exactness of the semidefinite and second-order cone relaxation of the OPF problem under normal operating conditions, cf. [1].

Returns

result – Data frame with the cycle analysis result for every subgrid, which comprises the following columns:

num_cycles: (hynet_int_) Number of cycles in the subgrid.

type: (BusType) Type of the subgrid.

buses: (pandas.Index) Pandas index with the *bus IDs* of the buses that are part of the subgrid.

Return type pandas.DataFrame

References

c_dst
Return the converter destination bus indices as a pandas series.

Remark: This property is *index-based* and intended for internal use.

c_src
Return the converter source bus indices as a pandas series.

Remark: This property is *index-based* and intended for internal use.

copy()
Return a deep copy of this scenario.

e_dst
Return the branch destination bus indices as a pandas series.

Remark: This property is *index-based* and intended for internal use.

e_src
Return the branch source bus indices as a pandas series.

Remark: This property is *index-based* and intended for internal use.

ensure_reference()
Automatically add a reference bus to all AC subgrids without a reference.

All AC subgrids are required to have a dedicated reference bus. However, cases may arise in which certain AC subgrids have no predefined reference, e.g., due to the partitioning or islanding of AC subgrids in a simulated branch outage. This method automatically assigns a reference bus to all AC subgrids without a reference, where the reference bus is set to the bus with the injector of highest capacity therein, if available.

get_ac_branches()
Return a pandas index with the branch IDs of all AC branches.

get_ac_subgrids()
Return a list with a pandas index of bus IDs for every AC subgrid.

get_branches_in_corridors(*corridors*)
Return a pandas index of branch IDs that reside in these corridors.

Remark: This property is *index-based* and intended for internal use.

Parameters **corridors** (*(numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_])*) – Tuple of NumPy arrays that state the source *bus index* and destination *bus index* of the corridors.

Returns **index** – Pandas index with the *branch IDs* of those branches that reside in the specified corridors.

Return type pandas.Index

get_bus_index(*bus_id*)
Return the bus index(es) for the given (iterable of) bus identifier(s).

Remark: This result is *index-based* and intended for internal use.

get_dc_branches()
Return a pandas index with the branch IDs of all DC branches.

get_dc_subgrids()
Return a list with a pandas index of bus IDs for every DC subgrid.

get_islands()
Return a list with a pandas index of bus IDs for every islanded grid.

get_parallel_branches()
Return a list with a pandas index of branch IDs for parallel branches.

get_ref_buses()
Return an array with the bus index of the reference buses.

Remark: This result is *index-based* and intended for internal use.

get_relative_loading()
Return the ratio of tot. active power load to tot. active power inj. cap.

This ratio indicates the relative amount of the active power injection capacity that is utilized by the active power load of this scenario.

get_time_string()
Return the scenario time stamp as `Dd HH:MM:SS`.

get_time_tuple()
Return the scenario time stamp as `(days, hours, minutes, seconds)`.

has_acyclic_ac_subgrids()
Return `True` if all AC subgrids are acyclic and `False` otherwise.

has_acyclic_dc_subgrids()
Return `True` if all DC subgrids are acyclic and `False` otherwise.

has_acyclic_subgrids()
Return `True` if all subgrids are acyclic and `False` otherwise.

n_src
Return the injector terminal bus indices as a pandas series.

Remark: This property is *index-based* and intended for internal use.

num_branches
Return the number of branches.

num_buses
Return the number of buses.

num_converters
Return the number of converters.

num_injectors
Return the number of injectors.

remove_buses(*buses*)
Remove the specified buses and attached branches, converters, and injectors.

Parameters **buses** (*Iterable[hynet_id_]*) – Iterable of bus IDs that specifies the buses to be removed.

Returns

- **branches** (*pandas.Index*) – Removed branches, which were connected to the removed buses.
- **converters** (*pandas.Index*) – Removed converters, which were connected to the removed buses.
- **injectors** (*pandas.Index*) – Removed injectors, which were connected to the removed buses.

set_conservative_rating()
Adjust the branch rating to a conservative setting (MATPOWER compat.).

In the *hynet* data format, the branch rating is an ampacity rating in terms of the apparent power flow at 1 p.u. (due to reasons stated in [1], Section III), i.e., the rating divided by `base_mva` is the ampacity rating. In the MATPOWER format, the rating is an apparent power rating, i.e., it is independent of the voltage. To ensure feasibility w.r.t. MATPOWER's apparent power rating, the *hynet* rating may be converted to a more conservative rating as described in [1], Remark 1, which is performed by this method.

References

set_minimum_series_resistance(*min_resistance*, *branch_ids=None*)
Set the series resistance of the branches to a minimum of `min_resistance`.

Very small values of the series resistance of branches may lead to numerical issues during the solution of the OPF problem. With this method, the series resistance is enforced to be larger or equal to `min_resistance`.

Parameters

- **min_resistance** (*float*) – Minimum resistance for the series resistance of the specified branches. If a branch has a series resistance below this value, it is replaced by `min_resistance`.
- **branch_ids** (*Iterable[hynet_id_]*) – Iterable of branch IDs for which the minimum series resistance shall be ensured. By default, all branches are considered.

verify(*log=<bound method Logger.warning of <Logger hynet.scenario.representation (WARNING)>>*)
Verify the integrity and validity of the scenario.

This method performs an extensive series of checks on the scenario to ensure that the data is consistent (e.g., the references between data frames), proper (e.g., constraint limits), and valid (i.e., compliant with all preconditions as assumed by *hynet*).

Parameters **log** (*function(str) or None*) – Function to log information about critical settings (default is the warning log of the module). Set to `None` to suppress this log output.

Raises **ValueError** – In case any kind of integrity or validity violation is detected.

verify_hybrid_architecture_conditions(*log=<bound method Logger.warning of <Logger hynet.scenario.representation (WARNING)>>*)
Return `True` if the *hybrid architecture*'s exactness results hold.

The *hybrid architecture* denotes a class of network topologies that, under very mild conditions, induces exactness to the semidefinite and second-order cone relaxation of the OPF problem in case that no *pathological price profile* emerges, see [1], [2], and [3]. This function returns `True` if the topological requirements of the *hybrid architecture* as well as the conditions on the system parameters that are utilized for the aforementioned results on exactness are satisfied for this scenario. It is assumed that the validity of the scenario is established beforehand, see `Scenario.verify`.

Parameters **log** (*function(str) or None*) – Function to log information about violated conditions (default is the warning log of the module). Set to `None` to suppress any log output.

References

See also:

hynet.scenario.representation.Scenario.verify()

3.1.8.5 hynet.scenario.verification module

Verification of a steady-state scenario.

hynet.scenario.verification.**verify_hybrid_architecture**(*scr*, *log_function*)
Return `True` if the *hybrid architecture*'s exactness results hold.

This function is actually part of the `Scenario` class. Due to its extent, it was moved to a separate module in order to improve code readability.

hynet.scenario.verification.**verify_scenario**(*scr*, *log_function*)
Verify the integrity and validity of the scenario.

This function is actually part of the `Scenario` class. Due to its extent, it was moved to a separate module in order to improve code readability.

Raises **`ValueError`** – In case any kind of integrity or validity violation is detected.

3.1.8.6 Module contents

Representation of a steady-state scenario in *hynet*.

3.1.9 hynet.solver package

3.1.9.1 Submodules

3.1.9.2 hynet.solver.cplex module

3.1.9.3 hynet.solver.cvxpy module

3.1.9.4 hynet.solver.ipopt module

3.1.9.5 hynet.solver.mosek module

3.1.9.6 hynet.solver.picos module

3.1.9.7 hynet.solver.pyomo module

3.1.9.8 Module contents

Solvers for the *hynet*-specific QCQP problem.

3.1.10 hynet.system package

3.1.10.1 Submodules

3.1.10.2 hynet.system.calc module

Calculate the solution of an optimization problem specified by a given model.

hynet.system.calc.**calc**(*model*, *solver=None*, *solver_type=<SolverType.QCQP: 'QCQP'>*, *initial_point_generator=None*)
Calculate the solution of the optimization problem for the given model.

Model classes take the scenario data to build a specific optimization problem. This function takes the model to generate the specification of the optimization problem, solve the problem using the specified solver, and route the result data though the model object to generate and return an appropriate result object. The solver or solver type may be specified explicitly, otherwise an appropriate solver is selected automatically.

Parameters

- **`model`** (`SystemModel`) – Model object that generates the quadratically constrained quadratic problem (QCQP).
- **`solver`** (`SolverInterface, optional`) – Solver for the QCQP problem; the default automatically selects an appropriate solver of the specified solver type.
- **`solver_type`** (`SolverType, optional`) – Solver type for the automatic solver selection (default `SolverType.QCQP`). It is ignored if `solver` is not `None`.

- **initial_point_generator** (*InitialPointGenerator or None, optional*) – Initial point generator for QCQP solvers (ignored for relaxation-based solvers). If `None` (default), the initial point generation is skipped.

Returns **result** – Result data of the solution of the optimization problem.

Return type *SystemResult*

See also:

hynet.system.model.SystemModel(), *hynet.system.result.SystemResult()*

hynet.system.initial_point() Initial point generators.

hynet.system.calc.**select_solver**(*solver_type*)
Return the most appropriate installed solver of the specified solver type.

Parameters **solver_type** (`SolverType`) – Specification of the solver type.

Returns **solver** – Selected solver interface *class* of the specified solver type.

Return type *SolverInterface*

Raises **RuntimeError** – In case no appropriate solver was found.

3.1.10.3 hynet.system.initial_point module

Initial point generation for the QCQP solvers for the OPF problem.

class hynet.system.initial_point.**InitialPointGenerator**
Bases: `abc.ABC`

Abstract base class for initial point generators for the QCQP OPF solvers.

Derived classes implement the generation of an initial point for solvers that solve the nonconvex QCQP representation of the OPF problem. With the provision of an appropriate initial point, the convergence performance and "quality" of the identified local optimum may be improved.

class hynet.system.initial_point.**RelaxationInitialPointGenerator**(*solver*, *rec_mse_thres=1e-08*)
Bases: *hynet.system.initial_point.InitialPointGenerator*

Relaxation-based initial point generator for QCQP solvers.

This generator returns an initial point for the solution of the QCQP that corresponds to a relaxation of the QCQP. Especially the second-order cone relaxation (SOCR solvers) is typically fast to compute and may be suitable.

solver
Return the solver for the initial point computation.

3.1.10.4 hynet.system.model module

Steady-state system model of *hynet*.

class hynet.system.model.**SystemModel**(*scenario*, *verify_scenario=True*)
Bases: `abc.ABC`

System model for a steady-state scenario of a grid.

Based on the specification of a scenario via a `Scenario` object, this class provides the methods to generate the corresponding system model equations and constraints. The state variables in this system model are the bus voltage vector `v`, the converter state vector `f`, the injector state vector `s`, and the auxiliary variable vector `z`. The latter is actually not part of the system model serves for auxiliary purposes, e.g., for the reformulation of the piecewise linear active and reactive power cost functions of injectors. All state variables are considered in p.u., i.e., normalized.

This class is designed as an abstract base class for specific problem formulations, like the optimal power flow problem, and serves as a builder for the optimization problem and as a factory for the associated result objects. Central to this process are the following member functions:

- **get_problem:** Returns the associated optimization problem as a quadratically constrained quadratic problem (QCQP). The construction of this QCQP object is defined by the following member functions:

  (a) `_get_objective` Returns the objective function object.

  (b) `_get_constraint_generators` Returns the constraint generation functions.

  (c) `get_v_bounds`: Returns the bounds on the magnitudes of the elements of the bus voltage vector `v`.

  (d) `get_f_bounds`: Returns the bounds on the elements of the converter state vector `f`.

  (e) `get_s_bounds`: Returns the bounds on the elements of the injector state vector `s`.

  (f) `get_z_bounds`: Returns the bounds on the elements of the auxiliary variable vector `z`.

  The methods (a) and (b) are abstract and must be implemented in a derived class to specify the optimization problem. While (c) - (e) are part of the system model and should not need any customization, (f) may be overridden. By default, the auxiliary variables are used for the reformulation of the piecewise linear active and reactive power cost functions of injectors into epigraph form as implemented in the method `get_cost_epigraph_constraints`. To customize the use of the auxiliary variables, `dim_z` and `get_z_bounds` (as well as `get_normalization_factors`) may be overridden in a derived class.

  Please note that the coefficient matrices of the constraints for the quadratic expressions in `v` must exhibit a sparsity pattern that corresponds to the network graph of the grid. If this is not fulfilled by a derived class, `get_problem` must be overridden accordingly.

- **create_result:** This method serves as a factory for an object that appropriately represents an optimization result for the model. It is abstract and must be implemented in a derived class.

See also:

hynet.scenario.representation.Scenario Specification of a steady-state grid scenario.

hynet.system.result.SystemResult Result of a system-model-related optimization.

hynet.system.calc.calc Calculate the solution of the optimization problem of a model.

Y
Return the bus admittance matrix.

Y_dst
Return the destination admittance matrix.

Y_src
Return the source admittance matrix.

calc_branch_angle_difference(*operating_point*)
Return the voltage angle difference along the branches in degrees.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

calc_branch_effective_rating(*operating_point*)
Return the ampacity rating in MVA rating at the current bus voltages.

In the scenario data, the branch flow is limited by an ampacity rating that is specified in terms of a long-term MVA rating at a bus voltage of 1 p.u.. Correspondingly, the ampacity rating translates to a different MVA rating if the bus voltages differ from 1 p.u.. This function returns the ampacity rating in MVA at the current bus voltages, i.e., the rating at 1 p.u. times the actual voltage.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

calc_branch_flow(*operating_point*)
Return the flow on the branches for the given operating point.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

Returns

- **i_src** (*numpy.ndarray[.hynet_complex_]*) – Branch current flow in p.u. at the source bus.
- **i_dst** (*numpy.ndarray[.hynet_complex_]*) – Branch current flow in p.u. at the destination bus.
- **s_src** (*numpy.ndarray[.hynet_complex_]*) – Branch apparent power flow in MVA at the source bus.
- **s_dst** (*numpy.ndarray[.hynet_complex_]*) – Branch apparent power flow in MVA at the destination bus.

calc_branch_voltage_drop(*operating_point*)
Return the relative voltage magnitude drop along the branches.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

calc_converter_flow(*operating_point*)
Return the apparent power flow into the converters.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

Returns

- **s_src** (*numpy.ndarray[.hynet_complex_]*) – Apparent power flow in MVA into the converter at the source bus.
- **s_dst** (*numpy.ndarray[.hynet_complex_]*) – Apparent power flow in MVA into the converter at the destination bus.

calc_converter_loss_error(*operating_point*)
Return the converter loss error.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

Returns **loss_err** – Loss error in MW due to noncomplementary modes of the converter.

Return type numpy.ndarray[hynet_float_]

calc_dynamic_losses(*operating_point*)
Return the total dynamic losses in MW for the given operating point.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

calc_injection_costs(*operating_point*)
Return the injector's active and reactive power injection costs in dollars.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

Returns

- **cost_p** (*numpy.ndarray[.hynet_float_]*) – Cost of the active power injection in dollars.
- **cost_q** (*numpy.ndarray[.hynet_float_]*) – Cost of the reactive power injection in dollars.

calc_shunt_apparent_power(*operating_point*)
Return the shunt apparent power in MVA for the given operating point.

Parameters **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

converter_loss_error_tolerance
Return the converter loss error tolerance in MW.

cost_function_scaling
Return the cost function scaling used by `get_cost_epigraph_constraints`.

To improve the numerical conditioning of the an optimization problem with the injector cost functions in the objective and, therewith, mitigate numerical issues with the solver, the cost functions may be scaled. This scaling must be considered when including other terms in the objective, e.g., a loss penalty.

create_result(*qcqp_result*, *total_time=nan*, *qcqp_result_pre=None*)
Create and return a result object associated with this model.

This method serves as a factory for a result object. Implement this method in a derived class to return an object of an appropriately customized result class.

Parameters

- **qcqp_result** (QCQPResult) – Solution of the QCQP associated with this model.
- **total_time** (*hynet_float_, optional*) – Total time for solving the optimization problem, cf. `hynet.system.calc.calc`.
- **qcqp_result_pre** (QCQPResult, *optional*) – Pre-solution of the model's QCQP for converter mode detection.

Returns **result**

Return type *hynet.system.result.SystemResult*

dim_f
Return the dimension of the state variable `f`.

dim_s
Return the dimension of the state variable `s`.

dim_v
Return the dimension of the state variable `v`.

dim_z
Return the dimension of the state variable `z`.

fix_converter_modes(*qcqp*, *operating_point*, *set_initial_point=True*)
Fix the converter modes in the QCQP according to the converter net flow.

This function determines the active converter mode based on the converter net active power flow in the provided operating point and updates the upper bounds on the converter state variable in the QCQP to fix the converter mode accordingly. Therewith, the emergence of any converter loss errors is suppressed.

Parameters

- **qcqp** (QCQP) – QCQP specification for the optimization problem associated with this model, see `get_problem`.
- **operating_point** (QCQPPoint) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result. Generally, this is the initial solution of the model's QCQP in which a nonnegligible converter loss error appeared.

- **set_initial_point** (*bool, optional*) – If `True` (default), the provided operating point is set as the initial point for the QCQP with the converter state variables adjusted according to the net active power flow.

get_angle_constraints()
Return the angle diff. lower and upper bound and "real-part" constraints.

get_balance_constraints()
Return the active and reactive power balance constraints.

get_converter_polyhedron_constraints()
Return the converter capability region polyhedron constraints.

get_cost_epigraph_constraints()
Return the PWL cost function epigraph constraints.

get_destination_ampacity_constraints()
Return the destination ampacity constraints.

get_drop_constraints()
Return the voltage drop lower and upper bound constraints.

get_dyn_loss_function()
Return `(C, c)` for the total dyn. loss `L(v,f) = v^H C v + c^T f`.

get_f_bounds()
Return the converter state bounds `f_lb <= f <= f_ub`.

get_injector_polyhedron_constraints()
Return the injector capability region polyhedron constraints.

get_normalization_factors()
Return the normalization factors of the state variables.

Returns **factors** – Contains the normalization factors of the state variables in the attributes `v`, `f`, `s`, and `z`.

Return type *QCQPPoint*

get_problem()
Return the optimization problem that is associated with this model.

Returns **qcqp** – QCQP specification for the optimization problem associated with this model.

Return type *QCQP*

get_real_part_constraints()
Return the "real-part" constraints.

These constraints ensure a precondition assumed by the voltage angle difference constraints, i.e., that the voltage angle difference is limited to +/- 90 degrees.

get_s_bounds()
Return the injector state bounds `s_lb <= s <= s_ub`.

get_source_ampacity_constraints()
Return the source ampacity constraints.

get_v_bounds()
Return the voltage magnitude bounds `v_lb <= |v| <= v_ub`.

Remark: The voltage magnitude bounds are captured by `get_voltage_constraints` and, thus, this box constraint is actually redundant and only included to provide optimization variable bounds for the solver (which, for some solvers, can improve convergence). To avoid any impact on the dual variables of the voltage magnitude constraints (which e.g. was observed with MOSEK), these box constraints are loosened w.r.t. the limits employed in `get_voltage_constraints`.

get_voltage_constraints()
Return the voltage magnitude lower and upper bound constraints.

get_z_bounds()
Return the auxiliary variable bounds `z_lb <= z <= z_ub`.

The lower and upper bound is set to `numpy.nan` if the corresponding bound should be omitted.

nodal_balance_error_tolerance
Return the relative nodal power balance error tolerance.

Threshold on the ratio of the maximum nodal apparent power balance error and the maximum individual apparent power load to consider a solution as physically valid.

scenario
Return the scenario data of the system model.

total_balance_error_tolerance
Return the relative total power balance error tolerance.

Threshold on the ratio of the total active power balance error and the total active power load to consider a solution as physically valid.

verify_converter_loss_accuracy(*operating_point*)
Return `True` if the converter loss error is within the tolerance.

The converter loss error tolerance of the model is specified by the property `converter_loss_error_tolerance`.

Parameters **operating_point** (`QCQPPoint`) – Operating point of the system *without* the normalization of the state variables, i.e., as provided by a QCQP result.

verify_power_balance_accuracy(*bal_err*)
Return `True` if the power balance error is within the tolerance.

The power balance error tolerance of the model is specified by the properties `nodal_balance_error_tolerance` and `total_balance_error_tolerance`.

Parameters **bal_err** (*numpy.ndarray[hynet_complex_]*) – Power balance error in MVA at the individual buses.

3.1.10.5 hynet.system.result module

Representation of a system-model-related optimization result.

class hynet.system.result.**SystemResult**(*model*, *qcqp_result*, *total_time=nan*, *qcqp_result_pre=None*)

Bases: `object`

Result of a system-model-related optimization.

Parameters

- **model** (`SystemModel`) – Model for the processed optimization problem.
- **empty** (*bool*) – `True` if the object does not contain any result data and `False` otherwise.
- **solver** (`SolverInterface`) – Solver object by which the result was obtained.
- **solver_status** (`SolverStatus`) – Status reported by the solver.
- **solver_time** (*float*) – Duration of the call to the solver in seconds.
- **optimal_value** (*float*) – Optimal objective value or `numpy.nan` if the solver failed.

- **total_time** (*float or numpy.nan*) – Total time for the calculation, including the modeling, solving, and result assembly. If not provided, this time is set to `numpy.nan`.
- **reconstruction_mse** (*float*) – Unavailable if the result is empty and, otherwise, the mean squared error of the reconstructed bus voltages in case of a relaxation and `numpy.nan` otherwise.
- **bus** (*pandas.DataFrame, optional*) – Unavailable if the result is empty and, otherwise, a data frame with the bus result data, indexed by the *bus ID*, which comprises at least the following columns:

  v: (hynet_complex_) Bus voltage rms phasor (AC) or bus voltage magnitude (DC).

  s_shunt: (hynet_complex_) Shunt apparent power in MVA. The real part constitutes the shunt losses in MW and the *negated* imaginary part constitutes the reactive power *injection*.

  bal_err: (hynet_complex_) Power balance residual in MVA, i.e., the evaluation of the complex-valued power balance equation at the system state. Theoretically, this should be identical to zero, but due to a limited solver accuracy and/or inexactness of the relaxation it is only approximately zero. This residual supports the assessment of solution accuracy and validity.

  dv_bal_p: (hynet_float_) Dual variable or KKT multiplier of the active power balance constraint.

  dv_bal_q: (hynet_float_) Dual variable or KKT multiplier of the reactive power balance constraint.

- **branch** (*pandas.DataFrame, optional*) – Unavailable if the result is empty and, otherwise, a data frame with the branch result data, indexed by the *branch ID*, which comprises at least the following columns:

  s_src: (hynet_complex_) Apparent power flow in MVA at the source bus (measured as a flow *into* the branch).

  s_dst: (hynet_complex_) Apparent power flow in MVA at the destination bus (measured as a flow *into* the branch).

  i_src: (hynet_complex_) Current flow in p.u. at the source bus (measured as a flow *into* the branch).

  i_dst: (hynet_complex_) Current flow in p.u. at the destination bus (measured as a flow *into* the branch).

  v_drop: (hynet_float_) Relative voltage magnitude drop from the source bus to the destination bus.

  angle_diff: (hynet_float_) Bus voltage angle difference in degrees between the source and destination bus.

  effective_rating: (hynet_float_) Ampacity in terms of a long-term MVA rating at the *actual* bus voltage. If no rating is available, it is set to `numpy.nan`.

  rel_err: (hynet_float_) Branch-related relative reconstruction error $\kappa_k(V^\star)$ as defined in equation (24) in [1] in case of a relaxed QCQP or `numpy.nan` otherwise.

  dv_real_part: (hynet_float_) Dual variable or KKT multiplier of the +/-90 degrees constraint on the angle difference (cf. equation (27) in [2]) or `numpy.nan` if unavailable.

- **converter** (*pandas.DataFrame*) – Unavailable if the result is empty and, otherwise, a data frame with the converter result data, indexed by the *converter ID*, which comprises at least the following columns:

p_src: (**hynet_float_**) Active power flow in MW at the source bus *into the converter*.

p_dst: (**hynet_float_**) Active power flow in MW at the destination bus *into the converter*.

q_src: (**hynet_float_**) Reactive power injection in Mvar at the source bus *into the grid*.

q_dst: (**hynet_float_**) Reactive power injection in Mvar at the destination bus *into the grid*.

loss_err: (**hynet_float_**) Loss error in MW due to noncomplementary modes of the converter.

loss_err_pre: (**hynet_float_**) Only available if the QCQP was pre-solved to detect and fix the converter modes. Loss error in MW in the pre-solution due to noncomplementary modes of the converter.

dv_p_fwd_min: (**hynet_float_**) Dual variable or KKT multiplier of the lower bound on the converter's forward mode active power flow or `numpy.nan` if unavailable.

dv_p_fwd_max: (**hynet_float_**) Dual variable or KKT multiplier of the upper bound on the converter's forward mode active power flow or `numpy.nan` if unavailable.

dv_p_bwd_min: (**hynet_float_**) Dual variable or KKT multiplier of the lower bound on the converter's backward mode active power flow or `numpy.nan` if unavailable.

dv_p_bwd_max: (**hynet_float_**) Dual variable or KKT multiplier of the upper bound on the converter's backward mode active power flow or `numpy.nan` if unavailable.

dv_cap_src_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region at the source bus or `numpy.nan` if unavailable.

dv_cap_src_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region at the source bus or `numpy.nan` if unavailable.

dv_cap_dst_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region at the destination bus or `numpy.nan` if unavailable.

dv_cap_dst_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region at the destination bus or `numpy.nan` if unavailable.

- **injector** (*pandas.DataFrame*) – Unavailable if the result is empty and, otherwise, a data frame with the injector result data, indexed by the *injector ID*, which comprises at least the following columns:

  s: (**hynet_complex_**) Apparent power injection in MVA.

  cost_p: (**hynet_float_**) Cost of the active power injection in dollars or `numpy.nan` if no cost function was provided.

  cost_q: (**hynet_float_**) Cost of the reactive power injection in dollars or `numpy.nan` if no cost function was provided.

  dv_cap_p_min: (**hynet_float_**) Dual variable or KKT multiplier of the active power lower bound of the capability region or `numpy.nan` if unavailable.

  dv_cap_q_min: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power lower bound of the capability region or `numpy.nan` if unavailable.

dv_cap_p_max: (**hynet_float_**) Dual variable or KKT multiplier of the active power upper bound of the capability region or `numpy.nan` if unavailable.

dv_cap_q_max: (**hynet_float_**) Dual variable or KKT multiplier of the reactive power upper bound of the capability region or `numpy.nan` if unavailable.

References

details
Return a formatted string with details of the system's state.

The returned string contains a formatted table for all major entity types, which is hopefully mostly self-explaining. In the very left or right of a column, there may be an indicator:

Indicator	Meaning
`R`	Reference bus in the respective subgrid.
`=`	The bus is a DC bus. If there is no indicator, the bus is an AC bus.
`*`	A limit on the respective quantity is active.
`>`	The branch is highly loaded, i.e., the flow is 90% or more of the effective rating.
`T`	The branch is a transformer. If there is no indicator, the branch is a line/cable.

get_branch_utilization()
Return a pandas Series with the branch utilization.

Returns **branch_utilization** – Utilization of the branches as the ratio of the MVA branch flow over the effective rating or `numpy.nan` for unrated branches.

Return type pandas.Series

get_dynamic_losses()
Return the dynamic losses in MW.

get_total_injection_cost()
Return the total injection cost in $/h.

get_total_losses()
Return the total losses in MW.

The total losses comprise the dynamic losses and the static losses of the converters.

has_valid_converter_flows
Return `True` if the converter loss error is within the model's tolerance.

has_valid_power_balance
Return `True` if the power balance error is within the model's tolerance.

is_physical
Return `True` if the flow errors are within the model's tolerance.

See also:

SystemResult.has_valid_converter_flows, *SystemResult.has_valid_power_balance*

is_valid
Return `True` if the result is considered as valid.

For a result to be valid, the solved must have terminated with the solver status `SOLVED` and the converter loss error and the power balance error must be within the model's tolerance.

num_branches
Return the number of branches.

num_buses
Return the number of buses.

num_converters
Return the number of converters.

num_injectors
Return the number of injectors.

scenario
Return the scenario data of the system model.

hynet.system.result.**ensure_result_availability**(*func*)
Decorates a result evaluation function with a result data availability check.

Functions that evaluate the result data typically require a check if the result data is available. This decorator offers a convenient and unified way to augment a result evaluation function with such a check, where the function must take a `SystemResult`-based object as the first argument.

3.1.10.6 Module contents

Steady-state system model representations in *hynet*.

3.1.11 hynet.test package

3.1.11.1 Submodules

3.1.11.2 hynet.test.installation module

Regression test to verify the proper installation of *hynet* at the user.

hynet.test.installation.**test_installation**(*verbose=True*)
Verify the OPF solution of a test system for all available solvers.

Parameters **verbose** (*bool, optional*) – If True (default), the test is documented to the standard output.

Returns **success** – True if the test was passed, False otherwise.

Return type bool

3.1.11.3 hynet.test.regression module

Support for OPF regression tests.

class hynet.test.regression.**OPFVerificationData**
Bases: *hynet.test.regression.OPFVerificationData*

OPF solution verification data for regression tests.

This class contains the minimum amount of reference data to properly verify the outcome of an OPF calculation. It was introduced to compactly embed the reference solution for installation testing in *hynet*'s code - it's not too pleasing, but simple and robust ;)

static create(*result*)
Return a OPF verification data object for the given OPF result.

Parameters **result** (`hynet.opf.result.OPFResult`) – OPF result for which the OPF verification data shall be created.

Returns **reference** – OPF verification data.

Return type *OPFVerificationData*

hynet.test.regression.**verify_opf_result**(*result*, *reference*, *tolerance*)

Verify an OPF result versus a reference solution.

The OPF result is verified by comparing the OPF verification data to a reference solution by calculating the respective relative mean absolute errors and checking them against a tolerance threshold. As the bus voltage magnitudes are less rigid (in particular w.r.t. relaxations), the error threshold is relaxed for those comparisons.

Parameters

- **result** (OPFVerificationData) – OPF solution that shall be verified.
- **reference** (OPFVerificationData) – Reference OPF solution.
- **tolerance** (*hynet_float_*) – Tolerance on the relative mean absolute error of the individual result vectors.

Raises **ValueError** – In case any mismatch beyond the tolerance is detected.

3.1.11.4 hynet.test.system module

Artificial system for OPF regression tests.

This testing data is embedded as code to obtain simple and robust access to it.

3.1.11.5 Module contents

Testing functionality to verify the proper setup of *hynet* at the user.

3.1.12 hynet.utilities package

3.1.12.1 Submodules

3.1.12.2 hynet.utilities.base module

General utilities.

class hynet.utilities.base.**Timer**

Bases: object

Measure time since Timer object creation or time measurements.

interval()

Return the time in seconds since the last measurement or object creation.

For the first measurement, the time elapsed since object creation is returned. From the second measurement onwards, the time elapsed since the last measurement is returned.

reset()

Reset the timer.

total()

Return the time in seconds since object creation.

hynet.utilities.base.**create_dense_vector**(*i*, *data*, *n*, *dtype=<class 'numpy.complex128'>*)

Create n-dim. vector x with x[i] = data[i].

hynet.utilities.base.**create_sparse_diag_matrix**(*diagonal*, *dtype=<class 'numpy.complex128'>*)

Return a diagonal matrix.

hynet.utilities.base.**create_sparse_matrix**(*i*, *j*, *data*, *m*, *n*, *dtype=<class 'numpy.complex128'>*)
Return an m-by-n sparse matrix A with A[i[k], j[k]] = data[k].

hynet.utilities.base.**create_sparse_zero_matrix**(*m*, *n*, *dtype=<class 'numpy.float64'>*)
Return an m-by-n all-zeros matrix.

hynet.utilities.base.**partition_iterable**(*iterable*, *num_blocks*)
Partition the iterable into the specified number of consecutive blocks.

Parameters

- **iterable** (*Iterable*) – Iterable to be partitioned.
- **num_blocks** (*hynet_int_*) – Number of blocks into which the iterable shall be partitioned.

Yields **blocks** (*Iterable*) – Blocks of the iterable.

hynet.utilities.base.**truncate_with_ellipsis**(*string*, *max_len*)
Truncate the string with an ellipsis if its length exceeds max_len.

3.1.12.3 hynet.utilities.chordal module

3.1.12.4 hynet.utilities.cvxopt module

Utilities related to CVXOPT.

hynet.utilities.cvxopt.**cvxopt2scipy**(*matrix*, *nonzero_only=False*)
Return the CVXOPT sparse matrix as a SciPy sparse matrix.

Parameters

- **matrix** (*cvxopt.spmatrix*) – Matrix that shall be converted.
- **nonzero_only** (*bool, optional*) – If False (default), all entries are transferred, regardless of their value. If True, the data is checked and only nonzero entries are transferred.

Returns Input matrix as a SciPy sparse matrix.

Return type hynet_sparse_

hynet.utilities.cvxopt.**scipy2cvxopt**(*matrix*)
Return the SciPy sparse matrix as a CVXOPT sparse matrix.

Parameters **matrix** (*hynet_sparse_*) – Matrix that shall be converted.

Returns Input matrix in CVXOPT's sparse format.

Return type cvxopt.spmatrix

3.1.12.5 hynet.utilities.graph module

Graph-related utilities.

hynet.utilities.graph.**eliminate_parallel_edges**(*edges*)
Eliminates parallel edges from the graph and returns those edges.

The edges are considered undirected, i.e., and the source and destination node are interchangeable. In the returned edges, the source is set to the adjacent node with the higher node number.

hynet.utilities.graph.**get_adjacency_matrix**(*edges*, *num_nodes=None*, *weights=None*)
Return the adjacency matrix for an undirected graph specified by its edges.

Parameters

- **edges** (*(numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_])*) – Edges of the graph in terms of node number tuples.
- **num_nodes** (*int, optional*) – Number of nodes in the graph. By default, this is set to the maximum node number in `edges` plus one.
- **weights** (*numpy.ndarray[hynet_float_], optional*) – Edge weights for a *weighted* adjacency matrix. For parallel edges, the weights are accumulated.

Returns **A** – Adjacency matrix of the undirected graph.

Return type hynet_sparse_

hynet.utilities.graph.**get_graph_components**(*nodes*, *edges*, *roots=None*)

Return a list of connected components.

The returned list contains an element for every connected component, where the element constitutes an array with all nodes of that component. The first element of the array is the root node of the respective component.

hynet.utilities.graph.**get_laplacian_matrix**(*edges*, *num_nodes=None*)

Return the Laplacian matrix for an undirected graph specified by its edges.

Parameters

- **edges** (*(numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_])*) – Edges of the graph in terms of node number tuples.
- **num_nodes** (*int, optional*) – Number of nodes in the graph. By default, this is set to the maximum node number in `edges` plus one.

Returns **L** – Laplacian matrix of the undirected graph.

Return type hynet_sparse_

hynet.utilities.graph.**get_minimum_spanning_tree**(*edges*, *weights*)

Return the minimum spanning tree in terms of an edges tuple.

Parameters

- **edges** (*(numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_])*) – Edges of the graph in terms of node number tuples.
- **weights** (*numpy.ndarray[hynet_float_], optional*) – Edge weights. For parallel edges, the weights are accumulated.

Returns **mst_edges** – Edges of the minimum spanning tree in terms of node number tuples.

Return type (numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_])

hynet.utilities.graph.**get_num_spanning_trees**(*edges*)

Return the number of spanning trees for the given undirected graph.

The number of spanning trees is determined using Kirchhoff's matrix tree theorem, see e.g. [1].

Caution: The vertices must be numbered consecutively starting from zero. If the graph is not connected, the number of spanning trees is zero.

Remark: As SciPy does not inherently support the computation of the determinant of a sparse matrix, the Laplacian matrix is converted to a dense (NumPy) matrix for the computation of a cofactor of the Laplacian. Due to this, the performance is potentially poor for large graphs.

Parameters **edges** (*(numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_])*) – Edges of the graph in terms of node number tuples.

Returns Number of spanning trees of the undirected graph.

Return type numpy.float64

:raises OverflowError : If the computation results in an overflow.:

References

hynet.utilities.graph.**is_acyclic_component**(*nodes*, *edges*, *root*)
Return `True` if the root's component is acyclic and `False` otherwise.

hynet.utilities.graph.**is_acyclic_graph**(*nodes*, *edges*)
Return `True` if the graph is acyclic and `False` otherwise.

hynet.utilities.graph.**traverse_graph**(*nodes*, *edges*, *callback*, *roots=None*, *auto_root=True*)
Traverse the graph and run a callback at every node.

The graph is traversed in a depth-first fashion and, at every visited node, the provided callback function is called, which must exhibit the signature:

```
def callback(node, node_pre, cycle)
```

Therein, `node` is the currently visited node, `node_pre` is the previously visited node (or `numpy.nan` if the current node is the component's root, i.e., the traversal entered a new component), and `cycle` is `True` if the current node was already visited, i.e., the edge traversed from the previous to the current node closes a cycle. The callback function returns an abort flag, i.e., if the callback returns `True`, the traversal is aborted.

Remark: This function is most efficient if the nodes are numbered consecutively from zero to the number of nodes minus one. Negative node numbers are not supported.

Parameters

- **nodes** (*numpy.ndarray[hynet_int_]*) – Node numbers of the graph.
- **edges** ((*numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_]*)) – Edges of the graph in terms of node number tuples.
- **callback** (*function*) – Node visit callback function.
- **roots** (*numpy.ndarray[hynet_int_], optional*) – Numbers of the root nodes of the graph.
- **auto_root** (*bool, optional*) – If True (default), components without a root node are assigned an arbitrary node as its root.

3.1.12.6 hynet.utilities.rank1approx module

Rank-1 approximation of a partial Hermitian matrix.

hynet.utilities.rank1approx.**calc_rank1approx_mse**(*V*, *v*, *edges*)
Calculate the mean squared error for the rank-1 approximation.

Returns the mean of the squared error of the elements on the diagonal as well as those on sparsity pattern defined by `edges`.

hynet.utilities.rank1approx.**calc_rank1approx_rel_err**(*V*, *v*, *edges*)
Calculate the relative reconstruction error for all edges.

hynet.utilities.rank1approx.**get_armijo_step_size**(*f*, *x*, *grad_fx*, *gamma=1*, *epsilon=0.2*, *alpha=2*)
Return a step size based on the Armijo-Goldstein condition.

This step size is designed for a Wirtinger calculus based gradient descent method that minimizes the function `f(x)`, where `x in C^N` is the current candidate solution, `grad_fx` is the gradient vector w.r.t. `conj(x)` evaluated at `x`, and the step direction `d = -grad_fx`. This function returns a step size `s in R_(+)`, such that the algorithmic map reads `x -> x + s * d`.

hynet.utilities.rank1approx.**rank1approx_via_least_squares**(*V*, *edges*, *roots*, *grad_thres=1e-07*, *mse_rel_thres=0.002*, *max_iter=300*, *show_convergence_plot=False*)

Return a rank-1 approximation `vv^H` of `V` by minimizing the squared error.

The rank-1 approximation may be put as the following optimization problem:

```
minimize  ||P(vv^H - V)||_F^2
v in C^N
```

Therein, `P(X)` is the projection of the matrix `X` onto the sparsity pattern defined by `edges`, i.e., all diagonal elements and the off-diagonal elements `(edges[k][0], edges[k][1])` and `(edges[k][1], edges[k][0])` for `k = 0,...,K-1`, where `K` is the number of edges. `||.||_F` denotes the Frobenius norm. For `V >= 0` (psd) and `V != 0`, this problem is nonconvex, i.e., the objective is not convex over the entire `C^N`. (If `rank(V) = 1` and `V = xx^H`, then the objective is locally convex at `x`.)

This function finds a (local) optimizer of this problem using a Wirtinger calculus based gradient descent with an Armijo-Goldstein step size control, which is initialized with the vector `v` obtained from the graph traversal rank-1 approximation method. (During the iterations it is ensured that the least squares error decreases, otherwise the iterations are aborted with a warning. Consequently, in terms of the least squares error, this approximation is at least as accurate as the traversal-based approximation.)

Parameters

- **V** (*hynet_sparse_*) – Matrix that should be approximated by `vv^H`.
- **edges** ((*numpy.ndarray[hynet_int_], numpy.ndarray[hynet_int_]*)) – Specification of the sparsity pattern of the matrix `V`.
- **roots** (*numpy.ndarray[hynet_int_]*) – Root nodes for the graph components of the sparsity pattern. The absolute angle in `v` is adjusted such that the absolute angle at the root nodes is zero.
- **grad_thres** (*hynet_float_, optional*) – Absolute threshold on the 2-norm of the objective's gradient w.r.t. `v` divided by `N`. (Termination criterion on the first order condition for local optimality.)
- **mse_rel_thres** (*hynet_float_, optional*) – Threshold on the relative improvement of `||P(vv^H - V)||_F^2 / N` from the candidate solution `v` in iteration `i` to `v` in iteration `i + 1`. (Termination criterion on stalling progress.)
- **max_iter** (*hynet_int_, optional*) – Maximum number of iterations. (Fallback in case the other termination criteria are not met in a reasonable number of iterations.)
- **show_convergence_plot** (*bool, optional*) – If True, a plot is shown to illustrate the convergence behavior.

Returns **v** – Vector `v`, where `vv^H` approximates `V` on the sparsity pattern.

Return type numpy.ndarray

hynet.utilities.rank1approx.**rank1approx_via_traversal**(*V*, *edges*, *roots*)

Return a rank-1 approximation `vv^H` of `V` based on a graph traversal.

Assuming that `V = vv^H` (i.e., `rank(V) = 1`), the off-diagonal element in row `i` and column `j` is `V_ij = v_i*conj(v_j)`. Thus, `|v_j| = sqrt(V_jj)` and `arg(v_j) = arg(v_i) - arg(V_ij)`. This is utilized to recover `v` by setting its element's magnitude to the square root of the diagonal elements of `V` and reconstructing the angle of its elements by accumulating the angle differences from the respective root node of in the graph associated with `V`. The angle at the root node(s) is set to zero.

3.1.12.7 hynet.utilities.worker module

Management of worker processes in *hynet*.

param workers Worker manager created at import time for parallel processing within the *hynet* package.

type workers WorkerManager

class hynet.utilities.worker.**WorkerManager**
Bases: object

Manage operations on a pool of workers.

This class provides operations on a pool of worker processes in case that parallel processing in *hynet* is enabled and, otherwise, it performs the operations in the current process. The pool of workers is maintained lazily, i.e., it is created on demand.

See also:

hynet.config

close_pool()
Close the pool of worker processes.

is_multiprocessing
Return True if the operations utilize multiprocessing.

map(*func*, *iterable*, *show_progress=False*, *unit='it'*, ***kwds*)
Apply the function to every item (single argument) in the iterable.

Parameters

- **func** (*function*) – Function that shall be applied to the items in the iterable. The function object must support pickling.
- **iterable** (*iterable*) – Iterable containing the function arguments. An item in this iterable is the single argument passed to the function. The iterable must support `len(iterable)`.
- **show_progress** (*bool, optional*) – If `True` (default `False`), the progress is reported to the standard output.
- **unit** (*str, optional*) – Name of one unit of iteration for the progress visualization (default `it`).
- **kwds** (*dict, optional*) – If parallel processing is enabled, the keyword arguments are passed to the `map`-method of `multiprocessing.Pool`.

Returns **result** – List of the function result for every item in the iterable.

Return type list

See also:

`multiprocessing.Pool.map()`

num_workers
Return the number of worker processes.

These workers are only active if parallel processing is enabled.

See also:

hynet.config

starmap(*func*, *iterable*, ***kwds*)
Apply the function to every item (argument tuple) in the iterable.

Parameters

- **func** (*function*) – Function that shall be applied to the items in the iterable. The function object must support pickling.
- **iterable** (*iterable*) – Iterable containing the function arguments. An item in this iterable is a tuple of the arguments passed to the function.
- **kwds** (*dict, optional*) – If parallel processing is enabled, the keyword arguments are passed to the `starmap`-method of `multiprocessing.Pool`.

Returns **result** – List of the function result for every item in the iterable.

Return type list

See also:

`multiprocessing.Pool.starmap()`

hynet.utilities.worker.**pool_operation**(*method*)

Decorates a worker operation method with a pool state verification.

The worker manager maintains its pool of workers lazily, i.e., it is only created on demand. Due to this, every potential pool operation in the worker manager class has to verify and, if necessary, update the pool state prior to the operation itself. This decorator manages this task. In the worker operation method, the code only needs to adapt to the availability of the pool.

3.1.12.8 Module contents

Collection of utilities for *hynet*.

3.1.13 hynet.visual package

3.1.13.1 Subpackages

hynet.visual.capability package

Submodules

hynet.visual.capability.settings module

Settings view to edit parameters in the capability region visualizer.

class hynet.visual.capability.settings.**DescriptiveEntry**(*master=None*, *name='Label'*, *alignment='horizontal'*, *value=0.0*)

Bases: `tkinter.ttk.Frame`

Container that bundles a label with an input field.

VALIDATION_REGEX = re.compile('^[+-]?([0-9]*[.])?[0-9]+$')

error_state(*old_value=None*)

Set the foreground to red and return the callback to undo this change.

Parameters **old_value** (*float*) – If this value is set, then the value of the input component is set to this value when this callback is called.

Returns Callback to undo this change (set the foreground back to black).

Return type function

get_value()

Return the current value of the field.

Returns

Return type float

reset_to_normal_state(*old_value=None*)
Reset the element to the normal state (black font on white background).

Parameters **old_value** (*float*) – If this value is set, then the value of the input component is set to this value when this callback is called.

set_callback(*callback*)
Set a callback that listens for value changes.

Parameters **callback** (*function*) – The function that should be called when a value has changed.

set_value(*value*)
Set the value for the field.

Parameters **value** (*float or str*) –

class hynet.visual.capability.settings.**PowerFactorView**(*master=None*)
Bases: tkinter.ttk.Frame

View for setting the power factor limit.

set_callback(*callback_function*)
Set the callback for updating the power factor limit.

Parameters **callback_function** (*function*) – Function that takes the power factor as an argument and updates the capability region.

class hynet.visual.capability.settings.**SettingsView**(*master=None, cap_region=[0.0,0.0]x[0.0,0.0]*)
Bases: tkinter.ttk.Frame

Settings view to edit parameters in the capability region visualizer.

deactivate_all_halfspaces()
Deactivate all checkboxes and, therewith, remove all half-spaces.

setup_halfspace_callbacks(*update_view_callback*)
Set the callback that updates the view with changes of the half-spaces.

Parameters **update_view_callback** (*function*) – This function is called when the checkbox is (de)activated and should update/reload the view of the capability region.

update_from_capability_region()
Update the views with the (possibly) changed capability region.

class hynet.visual.capability.settings.**ValueView**(*master=None, name='Unnamed ValueView', right_value_name='Slope:', left_value_name='Offset:', initial_value_tuple=(0, 0), checkbox_state=None*)
Bases: tkinter.ttk.Frame

Container view for two descriptive entries.

Furthermore, the value view handles the update of the associated half-space (if provided) as well as the error and update callbacks.

callback = None

update_from_capability_region(*halfspace*)
Updates this view based on the provided half-space specification.

Parameters **halfspace** (HalfSpace) – The half-space associated with this value view.

hynet.visual.capability.utilities module

This module contains various classes to provide interchangable data types between the view classes

class hynet.visual.capability.utilities.**AutoValue**
Bases: enum.Enum

This class provides a base class for enum that have a predefined, readable string representation.

class hynet.visual.capability.utilities.**Axis**
Bases: *hynet.visual.capability.utilities.AutoValue*

This enum represents the two axes.

ACTIVE_POWER = 1

REACTIVE_POWER = 2

class hynet.visual.capability.utilities.**Bound**
Bases: *hynet.visual.capability.utilities.AutoValue*

This enum represents the two bounds of the reactive and active limits of a capability region.

MAX = 2

MIN = 1

class hynet.visual.capability.utilities.**LinearFunction**(*halfspace*, *limits*)
Bases: object

Helper class to simplify the handling of the PWLs

f(*p*)

This function computes the PWL function for the input value p.

Parameters **p** (*float*) – The active power value for this linear function

Returns $ ext{self.slope}*p + ext{self.t}$

Return type The result of the linear curve equation for the input value

f_1(*q*)
This function computes the inverse PWL function for the input value q.

Parameters **q** (*float*) – The reactive power value for this linear function.

Returns $ (q - ext{self.t}) / ext{self.slope}$

Return type The result of the linear curve equation for the input value

intersect_with(*other*)
This function computes the intersection of this LinearFunction with the LinearFunction *other*.

Parameters **other** (LinearFunction) – The other LinearFunction we want to compute the intersection with.

Returns

- *Either None or a Point that represents the intersection of the two*
- *instances of LinearFunction*

class hynet.visual.capability.utilities.**Orientation**
Bases: *hynet.visual.capability.utilities.AutoValue*

This enum represents the four possibilities for halfspaces. These should only appear in pairs of two: {RIGHT, LEFT} x {TOP, BOTTOM}.

BOTTOM = 4

LEFT = 2

RIGHT = 1

TOP = 3

class hynet.visual.capability.utilities.**Point**
Bases: *hynet.visual.capability.utilities.Point*

Representation of a point in the P/Q-plane

hynet.visual.capability.visualizer module

Visualization of a capability region.

class hynet.visual.capability.visualizer.**Window**(*master=None, capability_region=[0.0,0.0]x[0.0,0.0], edit=True*)
Bases: tkinter.ttk.Frame

This class creates a window that visualizes a capability region.

CAPABILITY_REGION_COLOR = '#BBBBBB'

HALFSPACES_COLOR = '#777777'

POLYGON_EDGE_COLOR = '#777777'

POLYGON_FILL_COLOR = '#BAEBAC'

SCALING_FACTOR = 1

static compute_polygon_vertices(*cap_region*)
Compute the vertices of the capability region polygon.

This function computes the intersections of the four lines that delimit the capability region. The algorithm starts with the left-top half space and continues clock-wise with the other half-spaces.

The algorithm assumes that `p_max != p_min` and `q_max != q_min`. Furthermore, the assumptions on the slopes and offsets must be satisfied as well. If the preconditions are met, the function returns a list of all vertices that specify the polygon, in a clock-wise ordering.

Parameters **cap_region** (CapRegion) – The capability region that shall be drawn.

Returns List of polygon vertices.

Return type list[tuple(p,q)]

display_operating_point(*point*)
Display an operating point in the P/Q-plane of the capability region.

Parameters **point** (*tuple(p,q)*) – This tuple represents the operating point as a (p,q)-tuple of floats.

static show(*capability_region*, *edit=True*, *operating_point=None*)
Create and show a capability region visualizer window.

Parameters

- **capability_region** (CapRegion) –
- **edit** (*bool, optional*) – If True (default), editing of the specified capability region is enabled.
- **operating_point** (*tuple(p,q), optional*) – If provided, this operating point is shown by a marker in the P/Q-plane.

Module contents

3.1.13.2 Submodules

3.1.13.3 hynet.visual.graph module

Visualization of network graphs.

hynet uses NetworkX to export graph data. NetworkX can export to various popular formats, including the GraphViz DOT format and JSON.

hynet.visual.graph.**create_networkx_graph**(*data*)
Return a NetworkX graph object for the provided network graph.

Buses and injectors are inserted as nodes. Branches and converters are inserted as edges between the buses, while injectors are connected via additional edges to their respective terminal bus. The scenario data and, if provided, the OPF result data of the individual entities is added as attributes to the respective graph elements. The returned NetworkX graph can be exported into various formats, please refer to the documentation of NetworkX.

Parameters **data** (Scenario *or* SystemModel *or* SystemResult) – Scenario object, SystemModel object, or SystemResult object that contains the network graph information.

Returns **graph** – NetworkX graph representation of the grid's network graph.

Return type nx.Graph

See also:

hynet.visual.graph.export_networkx_graph_to_json()

hynet.visual.graph.**export_networkx_graph_to_json**(*graph*, *output_file*)
Exports a NetworkX graph object to a JSON file.

The specified network graph is exported to the JSON format, which is supported by D3 [1].

Remark: As a simple tool to visualize the graph, this subpackage includes the web page show_graph.html in the subdirectory rendering. To use it, store the graph object to the file network_graph.json and open the aforementioned HTML file.

Parameters

- **graph** (*nx.Graph*) – The graph that shall be exported to the JSON format.
- **output_file** (*str*) – The file name to which the JSON data shall be exported.

See also:

hynet.visual.graph.create_networkx_graph()

References

3.1.13.4 Module contents

Collection of visualization utilities for *hynet*.

3.2 Submodules

3.3 hynet.config module

hynet package configuration.

param GENERAL General settings.

parallelize: (bool) Enable or disable parallel processing in *hynet*. If True, certain procedures (e.g., the construction of constraint matrices for the OPF formulation) are parallelized if the system features more than one CPU.

type GENERAL dict

param OPF Optimal power flow settings.

pathological_price_profile_info: (bool) Enable or disable the output of information about pathological price profiles under the hybrid architecture in the OPF summary. See also `OPFResult` and `Scenario.verify_hybrid_architecture_conditions`.

type OPF dict

param DISTRIBUTED Settings for distributed computation.

default_port: (int) Default optimization server TCP port.

default_authkey: (str) Default optimization server authentication key.

default_num_workers: (int) Default number of worker processes on an optimization client.

ssh_command: (str) Command to run SSH on the local machine.

python_command: (str) Command to run Python on client machines.

type DISTRIBUTED dict

3.4 hynet.types_ module

Numeric types and enumerations in *hynet*.

class hynet.types_.**BranchType**
Bases: enum.Enum

Type of entity modeled by the branch.

LINE = 'line'

TRANSFORMER = 'transformer'

class hynet.types_.**BusType**
Bases: enum.Enum

Type of voltage waveform experienced at the bus.

AC = 'ac'

DC = 'dc'

class hynet.types_.**ConstraintType**
Bases: enum.Enum

Type of constraint for the QCQP specification.

EQUALITY = 'equality'

INEQUALITY = 'inequality'

class hynet.types_.**DBInfoKey**
Bases: enum.Enum

Valid keys for the `db_info` table in a *hynet* grid database.

BASE_MVA = 'base_mva'

DESCRIPTION = 'description'

GRID_NAME = 'grid_name'

VERSION = 'version'

class hynet.types_.**EntityType**
Bases: enum.Enum

Type of entity that is deactivated in a scenario.

BRANCH = 'branch'

BUS = 'bus'

CONVERTER = 'converter'

INJECTOR = 'injector'

SHUNT = 'shunt'

class hynet.types_.**InjectorType**
Bases: enum.Enum

Type of entity modeled by the injector.

BIOMASS = 'renewable:biomass'

COAL = 'conventional:coal'

COMPENSATION = 'compensation'

CONVENTIONAL = 'conventional'

GAS = 'conventional:gas'

GEOTHERMAL = 'renewable:geothermal'

HYDRO = 'renewable:hydro'

LOAD = 'load'

NUCLEAR = 'conventional:nuclear'

PROSUMER = 'prosumer'

PV = 'renewable:pv'

RENEWABLE = 'renewable'

WIND = 'renewable:wind'

is_compensation()
Return True if it is a reactive power compensator.

is_conventional()
Return True if it is a conventional generation utility.

is_load()
Return True if it is a load.

is_prosumer()
Return True if it is a prosumer.

is_renewable()
Return True if it is a renewables-based generation utility.

class hynet.types_.**SolverStatus**
Bases: enum.Enum

Status returned by a solver after performing an optimization.

FAILED = -4

INACCURATE = 1

INFEASIBLE = -2

SOLVED = 0

UNBOUNDED = -1

class hynet.types_.**SolverType**
Bases: enum.Enum

Type of problem that can be solved with the solver.

QCQP = 'QCQP'

SDR = 'SDR'

SOCR = 'SOCR'

3.5 Module contents

hynet: An optimal power flow framework for hybrid AC/DC power systems.

For more information, please refer to README.md, which is provided alongside *hynet*, as well as the docstrings of the individual classes and functions. The variables below are set up during package initialization.

param AVAILABLE_SOLVERS List of classes for all solvers available on the current system.

type AVAILABLE_SOLVERS list

param __version__ *hynet* version.

type __version__ str

CHAPTER 4

Change Log

All notable changes to this project are documented in this file.

4.1 [1.2.3] - 2021-05-02

- Updated the IPOPT solver interface to recent changes of the CyIpopt package.
- Fixed indexing issue in `convert_transformer_to_b2b_converter` and `convert_ac_line_to_hvdc_system`.
- Added a link to my dissertation to the README.
- Added a debug log output on the cause of solution failure for the PICOS solver interface.
- Changed the unit tests to use the IPOPT solver interface (instead of PICOS).

4.2 [1.2.2] - 2020-04-23

- Updated the PICOS solver interface to support PICOS v2.0.8.
- Updated the bibliography information for the publication of *hynet* in the IEEE Transactions on Power Systems.
- Updated some parts of the tutorials.
- Fixed issue with disappearing index names of the `add`-methods of `Scenario`.
- Removed the warning on negative load increments in `LoadabilityModel`.

4.3 [1.2.1] - 2019-08-29

- Fixed some compatibility issues with Matplotlib 3.1.1 in the visualization of the evaluation of the feature- and structure-preserving network reduction.
- Changed some data checks in the scenario verification that do not compromise compatibility from raising an exception to logging a warning.

4.4 [1.2.0] - 2019-08-19

- Revised the internal structure to simplify the implementation of extensions.
  - The former class `SystemModel` was split into the class `OPFModel` and an abstract base class `SystemModel`. The new `SystemModel` class implements the system model equations and serves as an interface class for optimization-problem-generating models. The class `OPFModel` specializes the new `SystemModel` to the formulation of the optimal power flow problem and replaces the former `SystemModel` class.
  - The class `OPFResult` was split into a specialization `OPFResult` and an abstract base class `SystemResult`. The `SystemResult` class implements a common framework for the representation and evaluation of a system model based optimization result. The new implementation of `OPFResult` specializes the `SystemResult` to the representation of an optimal power flow solution.
  - Some associated internal code revision and refactoring was performed. Changes that affect the user interface are documented below.
- Added an extension for the maximum loadability problem. Please refer to the tutorial "Maximum Loadability" for more information.
- Added a monitoring and automatic correction of the converter loss error: In problem formulations that incentivize a load increase at some buses, a loss error may emerge in lossy and bidirectional converters due to noncomplementary modes in the model. While this is typically not observed in OPF problems, it often emerges in the maximum loadability problem of hybrid systems. In the automatic correction, the converter mode is fixed according to the net active power flow in the initial solution and the problem is solved again, where a zero loss error is guaranteed.
- Added the properties `is_valid`, `has_valid_power_balance`, and `has_valid_converter_flows` to `SystemResult` (and `OPFResult`) and updated the property `is_physical` to improve and simplify the check of the result validity. Please refer to the updated tutorial "Analysis of the Optimal Power Flow Result" for more information.
- Added the methods `add_bus`, `add_branch`, `add_converter`, `add_injector`, and `get_relative_loading` to `Scenario`.
- Added the functions `convert_ac_line_to_hvdc_system` and `convert_transformer_to_b2b_converter`. Please refer to the tutorial "Construction of Hybrid AC/DC Grid Models" for more information.
- Added an argument to `Scenario.verify` to control its log output.
- Updated all internally issued scenario verifications to suppress any log output.
- Changed `Scenario.has_hybrid_architecture` to `Scenario.verify_hybrid_architecture_conditions`.
- Changed the attributes `total_injection_cost`, `dynamic_losses`, and `total_losses` of `SystemResult` (and `OPFResult`) to the methods `get_total_injection_cost`, `get_dynamic_losses`, and `get_total_losses`.
- Changed the columns `dv_cap_src_p_min`, `dv_cap_dst_p_min`, `dv_cap_src_p_max`, and `dv_cap_dst_p_max` of the `converter` data frame of `OPFResult` to `dv_p_fwd_min`, `dv_p_bwd_min`, `dv_p_fwd_max`, and `dv_p_bwd_max`, respectively.
- Changed the minimum version requirement for pandas from v0.23 to v0.24.
- Changed the default tolerance of the CPLEX SOCR solver interface to `5e-8`.
- Changed the default amalgamation of the MOSEK chordal SDR solver interface: The amalgamation improved the performance for many problem instances, but in some cases it led to numerical issues or less accurate results. In favor of robustness, the amalgamation is now disabled by default.
- Fixed some compatibility and deprecation issues with pandas v0.25 and NumPy v1.17.

- Removed `SystemModel.has_hybrid_architecture` and `SystemModel.islands`.

4.5 [1.1.4] - 2019-06-24

- Extended the OPF result summary for cases that are not solved successfully.
- Updated the MATPOWER import to support test cases with missing `gencost` data.
- Updated the MATPOWER import to detect and replace infinite (`Inf`) active/reactive power limits.
- Changed the automatic solver selection to *always* select a QCQP solver by default.
- Added `OPFResult.get_branch_utilization`.

4.6 [1.1.3] - 2019-06-13

- Revised the ampacity constraint generation to improve performance.

4.7 [1.1.2] - 2019-06-07

- Added a chordal conversion to the SDR solver interface for MOSEK.
- Added the suppression of the activity output of the clients in `OptimizationServer.start_clients`.
- Changed the progress bar of the `OptimizationServer` to `tqdm`.
- Updated the OPF summary (total losses in percent of the active power load).
- Updated the code to address the deprecation of `numpy.asscalar`.
- Updated the SOCR and SDR solver interface for MOSEK with a scaling of the coupling constraints for duplicate variables to improve the numerical accuracy of the duplication.
- Updated the SOCR solver interface for MOSEK to use a default of `1e-9` for `MSK_DPAR_INTPNT_CO_TOL_DFEAS` with versions prior to MOSEK v9.0.

4.8 [1.1.1] - 2019-05-17

- Added an IBM CPLEX based SOCR solver interface.
- Added an object-oriented design to the initial point generators and added their support in `calc_opf`.
- Updated the PICOS solver interface to support PICOS v1.2.0.
- Updated the MOSEK solver interface to support MOSEK v9.0.

4.9 [1.1.0] - 2019-03-28

- Added a feature- and structure-preserving network reduction method for large-scale grids.

4.10 [1.0.8] - 2019-02-26

- Added a setter for the grid name and description of a database (`DBConnection.grid_name` and `DBConnection.description`).
- Changed the default tolerance of the IPOPT QCQP solver to `1e-6` (was `1e-7`).

4.11 [1.0.7] - 2019-02-05

- Added average branch utilization statistics to the OPF summary.
- Added a local mode to the optimization server (replaces `num_local_workers`).
- Added a marginal price property to the `PWLFunction` class.
- Changed the automatic solver selection to require a QCQP solver for systems without the *hybrid architecture*.

4.12 [1.0.6] - 2019-01-10

- Fixed an issue in the MATPOWER import with optional data columns of the MATPOWER format.

4.13 [1.0.5] - 2019-01-10

- Added `Scenario.has_hybrid_architecture`, `Scenario.get_ac_branches`, `Scenario.get_dc_branches`, `Scenario.add_compensator`, `CapRegion.copy`, `show_power_balance_error`, and `show_branch_reconstruction_error`.
- Added an object-oriented design to the rank-1 approximation methods (to avoid the need of closures for their configuration).
- Added the detection of omitted ramping limits in the MATPOWER import.
- Extended the physical validity assessment that underlies `OPFResult.is_physical`.
- Updated the automatic solver selection and OPF result summary with the consideration of the *hybrid architecture*.
- Changed the default rank-1 approximation to the graph traversal method.
- Removed `SystemModel.is_acyclic`, `SystemModel.ac_subgrids`, and `SystemModel.dc_subgrids`.

4.14 [1.0.4] - 2018-12-28

- Revised the constraint scaling to improve performance.

4.15 [1.0.3] - 2018-12-11

- Extended the scenario verification to detect lines that connect buses with different base voltages.

4.16 [1.0.2] - 2018-12-07

- Revised the management of worker processes to improve performance, especially under Windows.

4.17 [1.0.1] - 2018-11-29

- Updated the README with solver installation instructions for Windows.
- Excluded support for CVXPY.

4.18 [1.0.0] - 2018-11-27

- Official release.

4.19 [0.9.9] - 2018-11-26

- Initial commit to GitLab.com.

4.20 [0.9.8] - 2018-10-19

- Pre-release of *hynet* on PyPI.

Bibliography

[1] C. Grigg et al., “The IEEE Reliability Test System - 1996. A report prepared by the Reliability Test System Task Force of the Application of Probability Methods Subcommittee,” in IEEE Trans. Power Systems, vol. 14, no. 3, pp. 1010-1020, Aug. 1999.

[1] https://github.com/pandas-dev/pandas/issues/13049

[1] CIGRE Working Group B2.41, “Guide to the Conversion of Existing AC Lines to DC Operation,” CIGRE Brochure 583, May 2014.

[1] G. D. Irisarri, X. Wang, J. Tong and S. Mokhtari, “Maximum loadability of power systems using interior point nonlinear optimization method,” IEEE Trans. Power Syst., vol. 12, no. 1, pp. 162-172, Feb. 1997.

[1] G. D. Irisarri, X. Wang, J. Tong and S. Mokhtari, “Maximum loadability of power systems using interior point nonlinear optimization method,” IEEE Trans. Power Syst., vol. 12, no. 1, pp. 162-172, Feb. 1997.

[1] G. D. Irisarri, X. Wang, J. Tong and S. Mokhtari, “Maximum loadability of power systems using interior point nonlinear optimization method,” IEEE Trans. Power Syst., vol. 12, no. 1, pp. 162-172, Feb. 1997.

[2] M. Hotz and W. Utschick, “The Hybrid Transmission Grid Architecture: Benefits in Nodal Pricing,” in IEEE Trans. Power Systems, vol. 33, no. 2, pp. 1431-1442, Mar. 2018.

[3] M. Hotz and W. Utschick, “A Hybrid Transmission Grid Architecture Enabling Efficient Optimal Power Flow,” in IEEE Trans. Power Systems, vol. 31, no. 6, pp. 4504-4516, Nov. 2016.

[1] M. Hotz and W. Utschick, “The Hybrid Transmission Grid Architecture: Benefits in Nodal Pricing,” in IEEE Trans. Power Systems, vol. 33, no. 2, pp. 1431-1442, Mar. 2018.

[2] M. Hotz and W. Utschick, “A Hybrid Transmission Grid Architecture Enabling Efficient Optimal Power Flow,” in IEEE Trans. Power Systems, vol. 31, no. 6, pp. 4504-4516, Nov. 2016.

[1] M. Hotz and W. Utschick, “The Hybrid Transmission Grid Architecture: Benefits in Nodal Pricing,” in IEEE Trans. Power Systems, vol. 33, no. 2, pp. 1431-1442, Mar. 2018.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] J. Sistermanns, M. Hotz, D. Hewes, R. Witzmann, and W. Utschick, “Feature- and Structure-Preserving Network Reduction for Large-Scale Transmission Grids,” 13th IEEE PES PowerTech Conference, Milano, Italy, Jun. 2019.

[1] M. Hotz and W. Utschick, “The Hybrid Transmission Grid Architecture: Benefits in Nodal Pricing,” in IEEE Trans. Power Systems, vol. 33, no. 2, pp. 1431-1442, Mar. 2018.

[1] M. Hotz and W. Utschick, “A Hybrid Transmission Grid Architecture Enabling Efficient Optimal Power Flow,” in IEEE Trans. Power Systems, vol. 31, no. 6, pp. 4504-4516, Nov. 2016.

[1] M. Hotz and W. Utschick, “A Hybrid Transmission Grid Architecture Enabling Efficient Optimal Power Flow,” in IEEE Trans. Power Systems, vol. 31, no. 6, pp. 4504-4516, Nov. 2016.

[2] M. Hotz and W. Utschick, “The Hybrid Transmission Grid Architecture: Benefits in Nodal Pricing,” in IEEE Trans. Power Systems, vol. 33, no. 2, pp. 1431-1442, Mar. 2018.

[3] M. Hotz and W. Utschick, “hynet: An Optimal Power Flow Framework for Hybrid AC/DC Power Systems,” in IEEE Trans. Power Systems, vol. 35, no. 2, pp. 1036-1047, Mar. 2020.

[1] M. Hotz and W. Utschick, “The Hybrid Transmission Grid Architecture: Benefits in Nodal Pricing,” in IEEE Trans. Power Systems, vol. 33, no. 2, pp. 1431-1442, Mar. 2018.

[2] M. Hotz and W. Utschick, "A Hybrid Transmission Grid Architecture Enabling Efficient Optimal Power Flow," in IEEE Trans. Power Systems, vol. 31, no. 6, pp. 4504-4516, Nov. 2016.

[1] Russell Merris, "Laplacian Matrices of Graphs: A Survey", Linear Algebra and its Applications, vol. 197-198, p. 143-176, 1994.

[1] https://github.com/d3/d3

Python Module Index

h

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z