pypowsybl

Author name not set

Apr 16, 2026

CONTENTS

PyPowSyBl is a python library for power grid modelling and simulation. It is a python binding for the powsybl java framework.

CHAPTER

ONE

GETTING STARTED

Install pypowsybl and start using it:

1.1 Getting started

1.1.1 Installation

You can simply install released versions of pypowsybl from PyPI using pip:

```
pip install pypowsybl
```

If you want to build pypowsybl from sources, please check out build instructions on github.

1.1.2 Basic usage

The main data type you will manipulate is the *Network*, defined in *pypowsybl.network* module.

In order to create one, you can use one of the provided creation methods, for example you can create the well known IEEE 9-bus test case:

```
import pypowsybl.network as pn
network = pn.create_ieee9()
```

You can then start inspecting the content of the network, for example in order to get lines data:

```
>>> network.get_lines()
       name     r      x   g1        b1   g2        b2  p1  q1  i1  p2  q2  i2 voltage_
↪level1_id voltage_level2_id bus1_id bus2_id  connected1  connected2
id
L7-8-0       0.85   7.20  0.0  0.000745  0.0  0.000745 NaN NaN NaN NaN NaN NaN          ␣
↪    VL2               VL8   VL2_1   VL8_0        True        True
L9-8-0       1.19  10.08  0.0  0.001045  0.0  0.001045 NaN NaN NaN NaN NaN NaN          ␣
↪    VL3               VL8   VL3_1   VL8_0        True        True
L7-5-0       3.20  16.10  0.0  0.001530  0.0  0.001530 NaN NaN NaN NaN NaN NaN          ␣
↪    VL2               VL5   VL2_1   VL5_0        True        True
L9-6-0       3.90  17.00  0.0  0.001790  0.0  0.001790 NaN NaN NaN NaN NaN NaN          ␣
↪    VL3               VL6   VL3_1   VL6_0        True        True
L5-4-0       1.00   8.50  0.0  0.000880  0.0  0.000880 NaN NaN NaN NaN NaN NaN          ␣
↪    VL5               VL1   VL5_0   VL1_1        True        True
L6-4-0       1.70   9.20  0.0  0.000790  0.0  0.000790 NaN NaN NaN NaN NaN NaN          ␣
↪    VL6               VL1   VL6_0   VL1_1        True        True
```

Each row of the obtained dataframe represents a line of the network.

You may have noticed that the flows are `NaN`. In order to compute them, you can run a loadflow using the *pypowsybl.loadflow* module:

```
import pypowsybl.loadflow as lf
lf.run_ac(network)
```

Flow values are now available in the lines dataframe:

```
>>> network.get_lines()[['p1', 'p2', 'q1', 'q2']].round(2)
           p1     p2     q1     q2
id
L7-8-0  76.38 -75.90  -0.80 -10.70
L9-8-0  24.18 -24.10   3.12 -24.30
L7-5-0  86.62 -84.32  -8.38 -11.31
L9-6-0  60.82 -59.46 -18.07 -13.46
L5-4-0 -40.68  40.94 -38.69  22.89
L6-4-0 -30.54  30.70 -16.54   1.03
```

You can also generate a single line diagram of one of the substations or voltage level, if you want to visualize the result:

```
>>> network.get_single_line_diagram('S1')
```

This will produce the following SVG image, which will just display if you run inside a notebook:

1.1.3 Going further

For more advanced use cases and topical guides, please check out the *User Guide*.

For a comprehensive list of features and detailed description of methods, please check out the *API reference*.

CHAPTER

TWO

USER GUIDE

Check out our topical user guides:

2.1 User Guide

The User Guide is meant to help the user on specific topics, relying as much as possible on practical examples.

2.1.1 The network model

The *Network* object is the main data structure of pypowsybl. It contains all the data of a power network: substations, generators, lines, transformers, ...

pypowsybl provides methods to create networks, and to access and modify their data.

Create a network

pypowsybl provides several factory methods to create well known network models. For example, you can create the IEEE 9-bus network case:

```
>>> network = pp.network.create_ieee9()
```

Another common way of creating a network is to load it from a file:

```
>>> network = pp.network.load('my-network.xiidm')
```

The supported formats are the following:

```
>>> pp.network.get_import_formats()
['BIIDM', 'CGMES', 'IEEE-CDF', 'JIIDM', 'MATPOWER', 'POWER-FACTORY', 'PSS/E', 'UCTE',
↪'XIIDM']
>>> pp.network.get_import_supported_extensions()
['RAW', 'RAWX', 'UCT', 'biidm', 'bin', 'dgs', 'iidm', 'jiidm', 'json', 'mat', 'raw',
↪'rawx', 'txt', 'uct', 'xiidm', 'xml']
```

Note

Import formats may support specific parameters, which you can find by using *get_import_parameters()*.

```
network = pp.network.load('ieee14.raw', {'psse.import.ignore-base-voltage': 'true'})
```

Loading a network from a binary byte buffer (io.BytesIO) is possible. Only zipped network loading are supported for now, but inside the zip file the supported network format are the same as pp.network.load.

```
with open('battery_xiidm.zip', "rb") as fh:
    n = pp.network.load_from_binary_buffer(io.BytesIO(fh.read()))
```

You may also create your own network from scratch, see below.

We can also configure some post processors to be loaded after import. To see the list of available post processors:

```
>>> pp.network.get_import_post_processors()
['geoJsonImporter', 'loadflowResultsCompletion', 'replaceTieLinesByLines']
```

Then a list of post processors can be pass to the load function:

```
network = pp.network.load('mycgmes.zip', post_processors=[
↪'replaceTieLinesByLines'])
```

Save a network

Networks can be written to the filesystem, using one of the available export formats:

```
network.save('network.xiidm', format='XIIDM')
```

You can also serialize networks to a string:

```
xiidm_str = network.save_to_string('XIIDM')
```

And also to a zip file as a (io.BytesIO) binary buffer.

```
zipped_xiidm = network.save_to_binary_buffer('XIIDM')
```

The supported formats are:

```
>>> pp.network.get_export_formats()
['AMPL', 'BIIDM', 'CGMES', 'JIIDM', 'MATPOWER', 'PSS/E', 'UCTE', 'XIIDM']
```

> **Note**
>
> Export formats may support specific parameters, which you can find by using *get_export_parameters()*.

Updating a network from a file

It is possible to only update part of a network data using the Network.update_from_file method (if the corresponding import format supports it). For example, you can import only an EQ file from a CGMES network, and then update it using the SSH file you need for the current situation.

The update can also be done using binary bytes buffers, using *Network.update_from_binary_buffer()* or *Network.update_from_binary_buffers()*

Reading network elements data

All network elements data can be read as `DataFrames`. Supported elements are:

- areas
- buses (from bus view)
- buses from bus/breaker view

- lines
- 2 windings transformers
- 3 windings transformers
- generators
- loads
- shunt compensators
- dangling lines
- LCC and VSC converters stations
- static var compensators
- switches
- voltage levels
- substations
- busbar sections
- HVDC lines
- ratio and phase tap changer steps associated to a 2 windings transformers
- identifiables that are all the equipment on the network
- injections
- branches (lines and two windings transformers)
- terminals are a practical view of those objects which are very important in the java implementation
- DC nodes
- DC lines
- voltage source converters
- DC grounds

Each element of the network is mapped to one row of the dataframe, an each element attribute is mapped to one column of the dataframe (a `Series`).

For example, you can retrieve generators data as follows:

```
>>> network = pp.network.create_eurostag_tutorial_example1_network()
>>> network.get_generators()
     name energy_source  target_p    min_p   max_p          min_q          max_q  rated_
↪s reactive_limits_kind  target_v  target_q  voltage_regulator_on regulated_element_id ␣
↪ p   q   i voltage_level_id   bus_id  connected
id
GEN               OTHER     607.0 -9999.99  4999.0  -9.999990e+03   9.999990e+03     ␣
↪NaN              MIN_MAX      24.5     301.0                  True                ␣
↪GEN NaN NaN NaN            VLGEN  VLGEN_0       True
GEN2              OTHER     607.0 -9999.99  4999.0 -1.797693e+308  1.797693e+308     ␣
↪NaN              MIN_MAX      24.5     301.0                  True                ␣
↪GEN2 NaN NaN NaN            VLGEN  VLGEN_0       True
```

Most dataframes are indexed on the ID of the elements. However, some more complex dataframes have a multi-index: for example, ratio and phase tap changer steps are indexed on their transformer ID together with the step position:

```
>>> network.get_ratio_tap_changer_steps()
                    side       rho    r    x    g    b
id         position
NHV2_NLOAD 0              0.850567  0.0  0.0  0.0  0.0
           1              1.000667  0.0  0.0  0.0  0.0
           2              1.150767  0.0  0.0  0.0  0.0
```

This allows to easily get steps related to just one transformer:

```
>>> network.get_ratio_tap_changer_steps().loc['NHV2_NLOAD']
          side       rho    r    x    g    b
position
0             0.850567  0.0  0.0  0.0  0.0
1             1.000667  0.0  0.0  0.0  0.0
2             1.150767  0.0  0.0  0.0  0.0
```

For a detailed description of each dataframe, please refer to the reference API *documentation*.

Updating network elements

Network elements can also be updated, using either simple values or list arguments, or `DataFrames` for more advanced cases. Not all attributes are candidates for update, for example element IDs cannot be updated. For a detailed description of what attributes can be updated please refer to the reference API *documentation*.

For example, to set the active power and reactive power of the load *LOAD*, the 3 following forms are equivalent:

- simple values as named arguments:

```
>>> network.update_loads(id='LOAD', p0=500, q0=300)
```

- lists or any sequence type as named arguments. Obviously this will be more useful if you need to update multiple elements at once. You must provide sequences with the same length (here 1):

```
>>> network.update_loads(id=['LOAD'], p0=[500], q0=[300])
```

- a full dataframe. This may be useful if you want to use data manipulation features offered by pandas library:

```
>>> df = pd.DataFrame(index=['LOAD'], columns=['p0','q0'], data=[[500, 300]])
>>> network.update_loads(df)
```

You can check that the load data has indeed been updated:

```
>>> network.get_loads()[['p0','q0']]
         p0     q0
id
LOAD  500.0  300.0
```

Basic topology changes

Most elements dataframes contain information about "is this element connected?" and "where is it connected?". That information appears as the `connected` and `bus_id` columns:

```
>>> network.get_generators()[['connected','bus_id']]
      connected   bus_id
id
```

(continues on next page)

(continued from previous page)

```
GEN         True  VLGEN_0
GEN2        True  VLGEN_0
```

You can disconnect or connect an element exactly the same way you update other attributes:

```
>>> network.update_generators(id='GEN', connected=False)
>>> network.get_generators()[['connected','bus_id']]
      connected   bus_id
id
GEN       False
GEN2       True  VLGEN_0
```

You can see that the generator *GEN* has been disconnected from its bus.

Working with multiple variants

You may want to change the state of the network while keeping in memory its initial state. In order to achieve that, you can use variants management:

After creation, a network has only one variant, called 'InitialState':

```
>>> network = pp.network.create_eurostag_tutorial_example1_network()
>>> network.get_variant_ids()
['InitialState']
```

You can then add more variants by cloning an existing variant:

```
>>> network.clone_variant('InitialState', 'Variant')
>>> network.get_variant_ids()
['InitialState', 'Variant']
```

You can then switch you "working" variant to the one you just created, and perform some operations on it, for example changing the target power of a generator to 700 MW:

```
>>> network.set_working_variant('Variant')
>>> network.update_generators(id='GEN', target_p=700)
>>> network.get_generators()['target_p']['GEN'].item()
700.0
```

If you switch back to the initial variant, you will see that its state has not changed, our generator still produces 607 MW:

```
>>> network.set_working_variant('InitialState')
>>> network.get_generators()['target_p']['GEN'].item()
607.0
```

We also provide an automatic way to switch to a variant and then go back to previous one with:

```
>>> with network.working_variant('Variant'):
...     network.update_generators(id='GEN', target_p=701)
...     network.get_generators()['target_p']['GEN'].item()
701.0
```

Once you're done working with your variant, you can remove it:

```
>>> network.remove_variant('Variant')
```

Create network elements

pypowsybl provides methods to add new elements (substations, lines, ...) to the network. This enables you to adapt an existing network, or even to create one from scratch.

As for updates, most creation methods accept arguments either as a dataframe or as named argument.

Let's create our network!

```
network = pp.network.create_empty()
```

First, we need to create some substations, let's create 2 of them:

```
network.create_substations(id=['S1', 'S2'])
```

Then, let's add some voltage levels inside those substations, this time with a dataframe:

```
voltage_levels = pd.DataFrame.from_records(index='id', data=[
    {'substation_id': 'S1', 'id': 'VL1', 'topology_kind': 'BUS_BREAKER', 'nominal_v':␣
↪400},
    {'substation_id': 'S2', 'id': 'VL2', 'topology_kind': 'BUS_BREAKER', 'nominal_v':␣
↪400},
])
network.create_voltage_levels(voltage_levels)
```

Let's now create some buses inside those voltage levels:

```
network.create_buses(id=['B1', 'B2'], voltage_level_id=['VL1', 'VL2'])
```

Let's connect these buses with a line:

```
network.create_lines(id='LINE', voltage_level1_id='VL1', bus1_id='B1',
                     voltage_level2_id='VL2', bus2_id='B2',
                     b1=0, b2=0, g1=0, g2=0, r=0.5, x=10)
```

Finally, let's add a load, and a generator to feed it through our line:

```
network.create_loads(id='LOAD', voltage_level_id='VL2', bus_id='B2', p0=100, q0=10)
network.create_generators(id='GEN', voltage_level_id='VL1', bus_id='B1',
                          min_p=0, max_p=200, target_p=100,
                          voltage_regulator_on=True, target_v=400)
```

You can now run a loadflow to check our network actually works !

```
>>> import pypowsybl.loadflow as lf
>>> res = lf.run_ac(network)
>>> str(res[0].status)
'ComponentStatus.CONVERGED'
```

Now let's see how to add a three-winding transformer to the network. First, let's add two voltage levels and their associated buses to the substation *S1*.

```
voltage_levels = pd.DataFrame.from_records(index='id', data=[
    {'substation_id': 'S1', 'id': 'VL3', 'topology_kind': 'BUS_BREAKER', 'nominal_v':␣
↪225},
    {'substation_id': 'S1', 'id': 'VL4', 'topology_kind': 'BUS_BREAKER', 'nominal_v': 90}
↪,
])
network.create_voltage_levels(voltage_levels)
network.create_buses(id=['B3', 'B4'], voltage_level_id=['VL3', 'VL4'])
```

Now let's add a three-winding transformer between VL1, VL2 and VL3:

```
network.create_3_windings_transformers(id='T1', rated_u0 = 225, voltage_level1_id='VL1',␣
↪bus1_id='B1',
                                       voltage_level2_id='VL3', bus2_id='B3',
                                       voltage_level3_id='VL4', bus3_id='B4',
                                       b1=1e-6, g1=1e-6, r1=0.5, x1=10, rated_u1=400,
                                       b2=1e-6, g2=1e-6, r2=0.5, x2=10, rated_u2=225,
                                       b3=1e-6, g3=1e-6, r3=0.5, x3=10, rated_u3=90)
```

You can add a ratio tap changer on the leg 1 of the three-winding transformer with:

```
rtc_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'target_deadband', 'target_v', 'oltc', 'low_tap', 'tap', 'side'],
    data=[('T1', 2, 200, False, 0, 1, 'ONE')])
steps_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'b', 'g', 'r', 'x', 'rho'],
    data=[('T1', 2, 2, 1, 1, 0.5),
          ('T1', 2, 2, 1, 1, 0.5),
          ('T1', 2, 2, 1, 1, 0.8)])
network.create_ratio_tap_changers(rtc_df, steps_df)
```

For more details and examples about network elements creations, please refer to the API reference *documentation*.

Creating node/breaker groups of elements

Pypowsybl provides ready for use methods to create a network with voltage levels in node/breaker topology from scratch or to modify an existing network in node/breaker topology. Before describing these methods, we are going to detailed how to do with elementary methods. Let's see how it works by creating a network from scratch. First, you need to create an empty network:

```
n = pp.network.create_empty()
```

Then you can add substations on the network. Below, two substations are added from a dataframe, called *S1* and *S2*.

```
stations = pd.DataFrame.from_records(index='id', data=[
   {'id': 'S1'},
   {'id': 'S2'}
])
n.create_substations(stations)
```

On each of these substations, you can create a node/breaker voltage level, here called *VL1* and *VL2*:

```
voltage_levels = pd.DataFrame.from_records(index='id', data=[
    {'substation_id': 'S1', 'id': 'VL1', 'topology_kind': 'NODE_BREAKER', 'nominal_v':␣
↪225},
    {'substation_id': 'S2', 'id': 'VL2', 'topology_kind': 'NODE_BREAKER', 'nominal_v':␣
↪225},
])
n.create_voltage_levels(voltage_levels)
```

Now, you can create busbar sections on every voltage level. Here, we create three busbar sections in voltage level *VL1* and one busbar section in voltage level *VL2*.

```
busbars = pd.DataFrame.from_records(index='id', data=[
    {'voltage_level_id': 'VL1', 'id': 'BBS1', 'node': 0},
    {'voltage_level_id': 'VL1', 'id': 'BBS2', 'node': 1},
    {'voltage_level_id': 'VL1', 'id': 'BBS3', 'node': 2},
    {'voltage_level_id': 'VL2', 'id': 'BBS4', 'node': 0},
])
n.create_busbar_sections(busbars)
```

You can draw single line diagrams for both the voltage levels to check that the busbar sections were added correctly. For more information about single line diagrams, check the *related documentation*.

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL2')
```

As you can see on the diagram of *VL1*, the busbar sections are not positioned in any chosen way. It is possible to add position extensions on these busbar sections to precise relative positions. Use *busbarSectionPosition* extension for that purpose. If you want to put the three busbar sections of *VL1* on the same slice, then they need to have the same *section_index*. As they belong to three distinct busbars, their *busbar_index* are different:

```
n.create_extensions('busbarSectionPosition', id='BBS1', busbar_index=1, section_index=1)
n.create_extensions('busbarSectionPosition', id='BBS2', busbar_index=2, section_index=1)
n.create_extensions('busbarSectionPosition', id='BBS3', busbar_index=3, section_index=1)
n.create_extensions('busbarSectionPosition', id='BBS4', busbar_index=1, section_index=1)
```

You can draw the single line diagram of *VL1* again to check that the busbar sections are at right positions.

```
>>> n.get_single_line_diagram('VL1')
```

Now let's connect a load in a close to reality way. The first thing to do is to create switches. To connect a load on *VL1*, you need to add a disconnector on each busbar sections of a slice, two open and one closed, and a breaker between the disconnectors and the load. You can add the switches with (note that the switch on *BBS1* is closed):

```
n.create_switches(pd.DataFrame.from_records(index='id', data=[
    {'voltage_level_id': 'VL1', 'id': 'DISC-BBS1-LOAD', 'kind': 'DISCONNECTOR', 'node1':␣
↪0, 'node2': 3, 'open':True},
    {'voltage_level_id': 'VL1', 'id': 'DISC-BBS2-LOAD', 'kind': 'DISCONNECTOR', 'node1':␣
↪1, 'node2': 3, 'open':False},
    {'voltage_level_id': 'VL1', 'id': 'DISC-BBS3-LOAD', 'kind': 'DISCONNECTOR', 'node1':␣
↪2, 'node2': 3, 'open':True},
    {'voltage_level_id': 'VL1', 'id': 'BREAKER-BBS1-LOAD', 'kind': 'BREAKER', 'node1': 3,
```

(continues on next page)

(continued from previous page)

```
↪ 'node2': 4, 'open':False},
]))
```

Now you can add the load on node 4:

```
n.create_loads(id='load1', voltage_level_id='VL1', node=4, p0=100, q0=10)
```

Now let's add a line between *VL1* and *VL2*. You need to add the switches first, three disconnectors and one breaker on *VL1* and one closed disconnector and one breaker on *VL2*:

```
n.create_switches(pd.DataFrame.from_records(index='id', data=[
    {'voltage_level_id': 'VL1', 'id': 'DISC-BBS1-LINE', 'kind': 'DISCONNECTOR', 'node1':␣
↪0, 'node2': 5, 'open':False},
    {'voltage_level_id': 'VL1', 'id': 'DISC-BBS2-LINE', 'kind': 'DISCONNECTOR', 'node1':␣
↪1, 'node2': 5, 'open':True},
    {'voltage_level_id': 'VL1', 'id': 'DISC-BBS3-LINE', 'kind': 'DISCONNECTOR', 'node1':␣
↪2, 'node2': 5, 'open':True},
    {'voltage_level_id': 'VL2', 'id': 'DISC-BBS4-LINE', 'kind': 'DISCONNECTOR', 'node1':␣
↪0, 'node2': 1, 'open':False},
    {'voltage_level_id': 'VL1', 'id': 'BREAKER-BBS1-LINE', 'kind': 'BREAKER', 'node1': 5,
↪ 'node2': 6, 'open':False},
    {'voltage_level_id': 'VL2', 'id': 'BREAKER-BBS4-LINE', 'kind': 'BREAKER', 'node1': 1,
↪ 'node2': 2, 'open':False},
]))
```

Then you can add the line with:

```
n.create_lines(id='line1', voltage_level1_id='VL1', voltage_level2_id='VL2', node1=6,␣
↪node2=2, r=0.1, x=1.0, g1=0, b1=1e-6, g2=0, b2=1e-6)
```

Now you can draw the single line diagrams of *VL1* and *VL2* to check that the load and the line have been correctly added:

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL2')
```

Here, similarly to busbar sections, the load and the line are randomly localized on the diagram. You can add extensions on the line and on the load to specify where they are localized in the busbar sections and if they must be drawn on the top or on the bottom. We choose that the load *load1* is the first feeder of the busbar section and on the bottom of *VL1*. The line *line1* is the second one and directed to the top on *VL1*. On *VL2*, the line is also on top. The relative position between the load and the line is specified with the order in the position extensions. The order of the load must be lower than the order of the line. You can use orders than do not follow each other to be able to add feeder later on.

```
n.create_extensions('position', id="load1", side="", order=10, feeder_name='load1',␣
↪direction='BOTTOM')
n.create_extensions('position', id=["line1", "line1"], side=['ONE', 'TWO'], order= [20,␣
↪10], feeder_name=['line1VL1', 'line1VL2'], direction=['TOP', 'TOP'])
```

Now you can draw the single line diagrams for both voltage levels again and see that the line and the load are now correctly positioned.

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL2')
```

Done but fastidious! That is why Pypowsybl provides ready-for-use methods to create an equipment and its bay with a single line. The switches are created implicitly. The methods take a busbar section on which the disconnector is closed as an argument (note that switches on the other parallel busbar sections are open). You also need to fill the position of the equipment on the voltage level as well as its characteristics. Optionally, you can indicate the direction of the equipment drawing - by default, on the bottom for injections and on top for lines and two windings transformers -, if an exception should be raised in case of problem - by default, True - and a report node to get logs.

You can add a load and connect it to *BBS3* between the line and the load1 (order position between 10 and 20) with:

```
pp.network.create_load_bay(n, id="load2", p0=10.0, q0=3.0, bus_or_busbar_section_id='BBS3
↪', position_order=15)
```

You can check that the load was added correctly by drawing a single line diagram of *VL1*:

```
>>> n.get_single_line_diagram('VL1')
```

Now let's connect a generator on *BBS1* on the left of *load1*, a dangling line on the right of *line1* on *BBS3* and a shunt on *BBS4*:

```
pp.network.create_generator_bay(n, id='generator1', max_p=1000, min_p=0, voltage_
↪regulator_on=True,
                                target_p=100, target_q=150, target_v=225, bus_or_busbar_
↪section_id='BBS1',
                                position_order=5)
pp.network.create_dangling_line_bay(n, id='dangling_line1', p0=100, q0=150, r=2, x=2,␣
↪g=1, b=1, position_order=30, bus_or_busbar_section_id='BBS3', direction='TOP')
shunt_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'model_type', 'section_count', 'target_v',
             'target_deadband', 'bus_or_busbar_section_id', 'position_order'],
    data=[('shunt1', 'LINEAR', 1, 221, 2, 'BBS4', 20)])
model_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'g_per_section', 'b_per_section', 'max_section_count'],
    data=[('shunt1', 0.014, 0.0001, 2)])
pp.network.create_shunt_compensator_bay(n, shunt_df=shunt_df, linear_model_df=model_df)
```

You can draw the new single line diagrams:

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL2')
```

These methods exist for every type of injections and also work with bus/breaker voltage levels. Then the injection is simply added to the given bus.

You can also add lines and two-windings transformers to a network with simple functions that create automatically the topology on both sides. For example, you can add a line and connect it between **BBS3** and **BBS4**, right next to line1

(order positions 25 on VL1 and 15 on VL2) with:

```
pp.network.create_line_bays(n, id='line2', r=0.1, x=10, g1=0, b1=0, g2=0, b2=0,
                                        bus_or_busbar_section_id_1='BBS3',
                                        position_order_1=25,
                                        direction_1='TOP',
                                        bus_or_busbar_section_id_2='BBS4',
                                        position_order_2=15,
                                        direction_2='TOP')
```

Here the direction of the line is specified in the argument, but it is optional and by default TOP. You can draw the single line diagrams of both voltage levels to check that the line was added correctly:

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL2')
```

Now let's see how to add a two windings transformer to our network. Both voltage levels VL1 and VL2 have a nominal voltage of 225kV. First, you need a new voltage level VL3, let's say of nominal voltage 63kV, to connect to one of the existing voltage level. Both voltage levels connected through a two-windings transformer must be in the same substation, let's pick VL1. You can create VL3 with:

```
n.create_voltage_levels(substation_id='S1', id='VL3', topology_kind='NODE_BREAKER',␣
↪nominal_v=63)
```

Now voltage level VL3 is created, but it is empty. The next step is to create the topology of the voltage level, i.e. the busbar sections and disconnectors or switches. You can use the same methods as to create VL1 and VL2, or use the built-in function allowing to create in one line the topology of symmetrical voltage levels. Let's create the topology with two busbar sections and three sections with breakers between sections one and two and disconnectors between sections two and three. You can do that with:

```
pp.network.create_voltage_level_topology(n, id='VL3', aligned_buses_or_busbar_count=2,␣
↪switch_kinds='BREAKER, DISCONNECTOR')
```

To check that the topology was correctly created, you can draw the single line diagram of voltage level VL3:

```
>>> n.get_single_line_diagram('VL3')
```

Now you can add some coupling devices between each section of the new voltage level. For that, you can use the built-in method and you just have to specify the two busbar sections on which the switches should be closed. Open switches will automatically be created on the parallel busbar sections:

```
pp.network.create_coupling_device(n, bus_or_busbar_section_id_1=['VL3_1_1', 'VL3_2_2'],␣
↪bus_or_busbar_section_id_2=['VL3_1_2', 'VL3_2_3'])
```

You can create the single line diagram to check that the coupling devices were well created:

```
>>> n.get_single_line_diagram('VL3')
```

Now, you can create a two windings transformer between VL1 and VL3. The features of the transformer must be specified, as well as the busbar sections on which it should be connected. You can connect it to BBS1 and to VL3_1_1. The position wanted for the transformer must be given for both ends and it is possible to indicate the direction for the diagram:

```
pp.network.create_2_windings_transformer_bays(n, id='two_windings_transformer', b=1e-6,␣
↪g=1e-6, r=0.5, x=10, rated_u1=225, rated_u2=63,
            bus_or_busbar_section_id_1='BBS1', position_order_1=35, direction_1='BOTTOM',
            bus_or_busbar_section_id_2='VL3_1_1', position_order_2=5, direction_2='TOP')
```

Let's draw the single line diagrams of VL1 and of VL3 to check that the two windings transformer is where it should be:

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL3')
```

To add a HVDC line to the network, you can first add the two converter stations, just like any other injection. Let's add one on the busbar section BBS3 of VL1 on the right and one on BBS4 on the right too:

```
pp.network.create_vsc_converter_station_bay(n, id=['VSC1', 'VSC2'], target_q=[200, 200],␣
↪voltage_regulator_on=[True, True], loss_factor=[1.0, 1.0],
            target_v=[230, 230], bus_or_busbar_section_id=['BBS3', 'BBS4'], position_
↪order=[30, 40])
```

Now you can add the HVDC line with:

```
n.create_hvdc_lines(id='HVDC_line', converter_station1_id='VSC1', converter_station2_id=
↪'VSC2',
                    r=1.0, nominal_v=400, converters_mode='SIDE_1_RECTIFIER_SIDE_2_
↪INVERTER',
                    max_p=1000, target_p=800)
```

The single line diagrams of voltage levels VL1 and VL2 are now:

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL2')
```

Now you know how to create a node-breaker voltage level and its topology, injections, lines and two-windings transformer with the built-in methods available in pypowsybl. For a reference of the available methods, please refer to: *documentation*.

Remove groups of elements

PyPowsybl provides build-in methods to remove existing elements such as feeders, voltage levels and HVDC lines. With these methods, it is possible to easily remove injections, lines and two windings transformers as well as the switches connecting them to a voltage level. Let's work on the network created in the section above.

You can remove the load1 with:

```
pp.network.remove_feeder_bays(n, 'load1')
```

The single line diagram of VL1 is then:

```
>>> n.get_single_line_diagram('VL1')
```

You can see that the load was removed, as well as all the breaker and disconnectors that was connecting it to the busbar section.

If you want to remove a HVDC line, you can use the built-in method that will remove not only the line but also the two converting stations and their switches.

You can remove HVDC_line with:

```
pp.network.remove_hvdc_lines(n, 'HVDC_line')
```

You can check on the single line diagram that everything went good:

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL2')
```

Finally, it is also possible to remove a full voltage level, with all its connectables. The lines and two windings transformers will be removed as well as their topology on both sides and the HVDC lines will be removed as well as their converter stations on both sides too.

For example, you can remove VL2 with:

```
pp.network.remove_voltage_levels(n, 'VL2')
```

The remaining voltage levels VL1 and VL3 are then:

```
>>> n.get_single_line_diagram('VL1')
```

```
>>> n.get_single_line_diagram('VL3')
```

On the diagrams, you can see that all the lines that were connecting VL1 to VL2 have been removed as well as their switches. On VL3, nothing was done as nothing was connected between VL2 and VL3.

Network merging

Pypowsybl provides methods to merge and detach networks. For example we can merge the 2 CGMES microgrids like this:

```
>>> be = pp.network.create_micro_grid_be_network()
>>> nl = pp.network.create_micro_grid_nl_network()
>>> be.merge(nl)
```

After the merge BE network has absorbed the NL network. So NL network is empty and BE network contains all the equipments that where in BE network before.

```
>>> nl.get_substations()
Empty DataFrame
Columns: [name, TSO, geo_tags, country]
Index: []

>>> be.get_substations()
                                          name TSO         geo_tags country
id
37e14a0f-5e34-4647-a062-8bfd9305fa9d   PP_Brussels        ELIA-Brussels      BE
```

(continues on next page)

(continued from previous page)

```
87f7002b-056f-4a6a-a872-1744eea757e3        Anvers          ELIA-Anvers      BE
c49942d6-8b01-4b01-b5e8-f1180f84906c  PP_Amsterdam      TENNET TSO B.V.      NL
```

Once merged, we can still keep track of original networks. They are called "sub networks" and can be listed from their parent networks like this:

```
>>> be.get_sub_networks()
Empty DataFrame
Columns: []
Index: [urn:uuid:d400c631-75a0-4c30-8aed-832b0d282e73, urn:uuid:77b55f87-fc1e-4046-9599-
↪6c6b4f991a86]
```

We can see that the sub network dataframe has not yet columns and contains two rows corresponding to the 2 BE and NL sub networks. We can also get a sub network from its parent network and then only work and focus on a particular sub network. When we get substations from the NL sub network we obviously only get one substation like in the original non merged network.

```
>>> nl_sub = be.get_sub_network('urn:uuid:77b55f87-fc1e-4046-9599-6c6b4f991a86')
>>> nl_sub.get_substations()
                                          name TSO         geo_tags country
id
c49942d6-8b01-4b01-b5e8-f1180f84906c  PP_Amsterdam      TENNET TSO B.V.      NL
```

Original networks can be recovered thanks to the "detach" method. In that case the sub network is removed from its parent network and become again a standalone network.

```
>>> nl_sub.detach()
```

Reducing a network

Pypowsybl provides methods to reduce a network to a smaller one. It can be done with different parameters. It can be decided according to the voltage with the parameters v_min and v_max. It can also be by indicating the Voltage Levels that will be kept and also indicating the depth around these voltage levels.

For this example we will keep only voltage levels with voltage superior or equal to 400 kV

```
>>> net = pp.network.create_four_substations_node_breaker_network()
>>> net.get_voltage_levels()
      name substation_id  nominal_v  high_voltage_limit  low_voltage_limit
id
S1VL1                 S1      225.0               240.0              220.0
S1VL2                 S1      400.0               440.0              390.0
S2VL1                 S2      400.0               440.0              390.0
S3VL1                 S3      400.0               440.0              390.0
S4VL1                 S4      400.0               440.0              390.0

>>> net.reduce_by_voltage_range(v_min=400)
>>> net.get_voltage_levels()
      name substation_id  nominal_v  high_voltage_limit  low_voltage_limit
id
S1VL2                 S1      400.0               440.0              390.0
S2VL1                 S2      400.0               440.0              390.0
S3VL1                 S3      400.0               440.0              390.0
S4VL1                 S4      400.0               440.0              390.0
```

For the next example we will keep voltage level S1VL1 with a depth of 1.

```
>>> net = pp.network.create_four_substations_node_breaker_network()
>>> net.get_voltage_levels()
      name substation_id  nominal_v  high_voltage_limit  low_voltage_limit
id
S1VL1                S1      225.0               240.0              220.0
S1VL2                S1      400.0               440.0              390.0
S2VL1                S2      400.0               440.0              390.0
S3VL1                S3      400.0               440.0              390.0
S4VL1                S4      400.0               440.0              390.0
>>> net.reduce_by_ids_and_depths(vl_depths=[('S1VL1', 1)])
>>> net.get_voltage_levels()
      name substation_id  nominal_v  high_voltage_limit  low_voltage_limit
id
S1VL1                S1      225.0               240.0              220.0
S1VL2                S1      400.0               440.0              390.0
```

S1VL1 is connected to S1VL2 by the transformer TWT, so it is kept after the network reduction. It is the only voltage level connected to S1VL1 by one branch.

Reduction can also be done by specifying directly the list of voltage levels to keep using the :meth:reduce_by_ids method.

Using operational limits

Operational limits can be added on various network elements : - each side of the branches (lines and 2 windings transformers) - each leg of 3 windings transformers - dangling lines

For more information on the model of operational limits, see the internal model documentation

Limits are defined in operational limit groups, that are represented by an *id*. Each line, transformer or dangling line is associated with a collection of groups, and for each element one of its groups is the selected one. The id of the selected limit group can be found in the data of each element :

```
>>> net = pp.network.create_eurostag_tutorial_example1_network()
>>> net.get_lines(attributes=["selected_limits_group_1", "selected_limits_group_2"])
           selected_limits_group_1 selected_limits_group_2
id
NHV1_NHV2_1                 DEFAULT                 DEFAULT
NHV1_NHV2_2                 DEFAULT                 DEFAULT
```

To retrieve the limits present on a network, the user can use the *get_operational_limits* method (choosing with the *show_inactive_sets* parameter whether to get all groups or only the active ones). For example on a network with a line that has two sets of current limits :

```
>>> net.get_operational_limits() # only selected sets

           element_type side             name     type     value  acceptable_duration
element_id
LINE1              LINE  TWO  permanent_limit  CURRENT      1000                   -1
LINE1              LINE  TWO              10'  CURRENT      1200                  600
LINE1              LINE  TWO               1'  CURRENT      1500                   60

>>> net.get_operational_limits(all_attributes=True, show_inactive_sets=True) # all sets
```

(continues on next page)

(continued from previous page)

```
            element_type side             name     type     value  acceptable_duration ␣
↪fictitious     group_name  selected
element_id
LINE1               LINE  TWO  permanent_limit  CURRENT      1000                   -1  ␣
↪    False        DEFAULT      True
LINE1               LINE  TWO              10'  CURRENT      1200                  600  ␣
↪    False        DEFAULT      True
LINE1               LINE  TWO               1'  CURRENT      1500                   60  ␣
↪    False        DEFAULT      True
LINE1               LINE  TWO  permanent_limit  CURRENT      1100                   -1  ␣
↪    False    OTHER_GROUP     False
```

Finally, the user can change the selected group of limits using the respective update methods of network elements. Using the same example as above :

```
>>> net.update_lines(id='LINE1', 'selected_limits_group_2'='OTHER_GROUP')
>>> net.get_operational_limits(all_attributes=True, show_inactive_sets=True)

            element_type side             name     type     value  acceptable_duration ␣
↪fictitious     group_name  selected
element_id
LINE1               LINE  TWO  permanent_limit  CURRENT      1000                   -1  ␣
↪    False        DEFAULT     False
LINE1               LINE  TWO              10'  CURRENT      1200                  600  ␣
↪    False        DEFAULT     False
LINE1               LINE  TWO               1'  CURRENT      1500                   60  ␣
↪    False        DEFAULT     False
LINE1               LINE  TWO  permanent_limit  CURRENT      1100                   -1  ␣
↪    False    OTHER_GROUP      True
```

2.1.2 Network visualization

Single line diagram

To create a single line diagram in SVG format from a substation or a voltage level:

```
>>> network = pp.network.create_ieee14()
>>> network.write_single_line_diagram_svg('VL4', 'vl4.svg')
```

Or in a Jupyter notebook, the SVG can be directly rendered in the notebook:

```
>>> network.get_single_line_diagram('VL4')
```

Note that a loadflow can be run before writing the diagram so that it displays reactive and active powers:

```
>>> network = pp.network.create_ieee14()
>>> result = pp.loadflow.run_ac(network)
>>> network.write_single_line_diagram_svg('VL4', 'vl4.svg')
```

Customizing with SldParameters

Single-line diagrams can be customized through SldParameters:

```
>>> network = pp.network.create_ieee14()
>>> result = pp.loadflow.run_ac(network)
>>> network.get_single_line_diagram('VL4',parameters = pp.network.SldParameters(use_name␣
↪= False, center_name = False, diagonal_label = False, nodes_infos = False, tooltip_
↪enabled = False, topological_coloring = True, component_library = 'Convergence'))
```

- use_name: if true, display components names instead of their id (default value false)
- center_name: if true, center the names of feeders (default value false)
- diagonal_label: if true, display labels diagonally (default value false)
- nodes_infos: if true, display nodes infos (caption on bottom left) (default value false)
- tooltip_enabled: if true, display the name of the component pointed by the cursor (default value false)
- topological_coloring: if true, set each electrical nodes with a different colour (default value true)
- component_library: choose component library (default value 'Convergence')
- active_power_unit: display unit of active power (default value "")
- reactive_power_unit: display unit of reactive power (default value "")
- current_unit: display unit of current (default value "")
- display_current_feeder_info: if true, display current feeder value (default value False)

Let's see some examples down below:

- with default parameters

```
>>> import pypowsybl.network as pn
>>> param = pn.SldParameters() #default parameters
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with use_name = True

```
>>> param = pn.SldParameters(use_name = True)
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with center_name = True

```
>>> param = pn.SldParameters(center_name = True)
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with diagonal_label = True

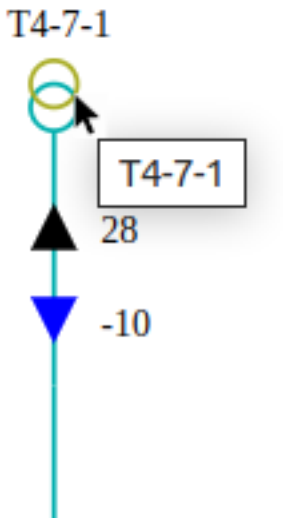
```
>>> param = pn.SldParameters(diagonal_label = True)
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with nodes_infos = True

```
>>> param = pn.SldParameters(nodes_infos = True)
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with tooltip_enabled = True

```
>>> param = pn.SldParameters(tooltip_enabled = True)
>>> network.get_single_line_diagram('VL4', parameters = param)
```



- with topological_coloring = True

```
>>> network = pn.create_four_substations_node_breaker_network()
>>> network.update_switches(id="S1VL2_COUPLER",open=True)
>>> param = pn.SldParameters(topological_coloring = True)
>>> network.get_single_line_diagram('S1VL2', parameters = param)
```

- with topological_coloring = False

```
>>> param = pn.SldParameters(topological_coloring = False)
>>> network.get_single_line_diagram('S1VL2', parameters = param)
```

- with component_library = “FlatDesign”

```
>>> network = pn.create_ieee14()
>>> param = pn.SldParameters(component_library = "FlatDesign")
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with display_current_feeder_info = True

```
>>> param = pn.SldParameters(display_current_feeder_info = True)
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with active_power_unit = “MW”

```
>>> param = pn.SldParameters(active_power_unit = "MW")
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with reactive_power_unit = "MVAR"

```
>>> param = pn.SldParameters(reactive_power_unit = "MVAR")
>>> network.get_single_line_diagram('VL4', parameters = param)
```

- with current_unit = "A"

```
>>> param = pn.SldParameters(display_current_feeder_info = True, current_unit = "A")
>>> network.get_single_line_diagram('VL4', parameters = param)
```

It is also possible to display a multi-substation single line diagram (currently a beta feature):

```
>>> network = pp.network.create_ieee14()
>>> result = pp.loadflow.run_ac(network)
>>> network.write_matrix_multi_substation_single_line_diagram_svg([['S1', 'S2'],['S3','S4
↪']], 's1_s2_s3_s4.svg')
```

Or in a Jupyter Notebook:

```
>>> network.get_matrix_multi_substation_single_line_diagram([['S1', 'S2'],['S3','S4']])
```

The substation diagrams will be arranged in a grid, based on the content of the matrix parameter. An empty string in the matrix will result in an empty spot in the grid.

Customizing with SldProfile

The single line diagram can be further customized using an SldProfile. For example, to set the labels for feeders and buses by using dataframes:

```
>>> import pypowsybl.network as pn
>>> import pandas as pd

>>> network = pn.create_ieee14()

>>> sld_labels_df = pd.DataFrame.from_records(index='id', columns=['id', 'label',
↪'additional_label'],
                                  data=[('B1-G', 'MY-GENERATOR', 'GEN'),
                                        ('L1-5-1', 'MY-LINE1', None),
                                        ('L1-2-1', 'MY-LINE2', None),
                                        ('B1', 'MY-BUS1', None)])

>>> sld_feeders_info_df = pd.DataFrame.from_records(index='id', columns=['id', 'type',
↪'side', 'direction', 'label'],
                                       data=[('L1-5-1', 'ARROW_ACTIVE', 'ONE', 'IN',
↪'ACTIVE1'),
                                             ('L1-5-1', 'ARROW_REACTIVE', 'ONE', 'OUT',
↪'REACTIVE1'),
                                             ('L1-2-1', 'ARROW_CURRENT', 'ONE', 'IN',
↪'CURRENT1')])

>>> sld_styles_df = pd.DataFrame.from_records(index='id', columns=['id', 'color', 'bus_
↪width', 'width', 'dash'],
                                  data=[('B1', 'orange', '4px', '2px', '')])
```

(continues on next page)

(continued from previous page)

```
>>> diagram_profile=pn.SldProfile(labels=sld_labels_df, feeders_info=sld_feeders_info_df,
↪ styles= sld_styles_df)
>>> network.get_single_line_diagram('VL1', sld_profile=diagram_profile)
```

In the labels dataframe:

- id: is the network element id
- label: defines the label for the element
- additional_label: defines an additional label, displayed on the right side of the element

In the feeders_info dataframe:

- id is the feeder id
- type: is a symbol type for the feeder info element. E.g., ARROW_ACTIVE, ARROW_REACTIVE, ARROW_CURRENT.
- side: for feeders with multiple sides (e.g. lines, transformers), determines at which side the feeder info is placed. E.g. ONE, TWO.
- direction: is direction of the arrows (IN, OUT).
- label: defines the label for the feeder info element..

In the styles dataframe:

- id is the bus id
- color: is the color for the bus and the elements connected to the bus
- bus_width: is the width for the bus
- width: is the width for the elements connected to the bus
- dash: is a string that specifies the lengths of alternating dashes and gaps, separated by commas and/or spaces

The optional parameter sld_profile can also be set for the write_single_line_diagram_svg, get_matrix_multi_substation_single_line_diagram, and write_matrix_multi_substation_single_line_diagram_svg functions.

Network area diagram

To create a network area diagram in SVG format for the full network:

```
>>> network = pp.network.create_ieee9()
>>> network.write_network_area_diagram_svg('ieee9.svg')
```

Or in a Jupyter notebook, the SVG can be directly rendered in the notebook:

```
>>> network.get_network_area_diagram()
```

To render only a part of the network, we can specify a voltage level ID as the center of the sub network and a depth to control the size of the sub network:

```
>>> network = pp.network.create_ieee300()
>>> network.write_network_area_diagram_svg('ieee300.svg', 'VL1', 1)
```

Nominal voltage bounds can be defined to further filter the output network:

```
>>> network = pp.network.create_ieee300()
>>> network.write_network_area_diagram_svg('ieee300.svg', 'VL1', 1, low_nominal_voltage_
↪bound=90, high_nominal_voltage_bound=240)
```

If no voltage level ID is given as an input, only nominal voltage bounds are used to filter the network:

```
>>> network = pp.network.create_ieee30()
>>> network.write_network_area_diagram_svg('ieee30.svg', low_nominal_voltage_bound=90,␣
↪high_nominal_voltage_bound=240)
```

Note that similarly to single-line diagrams, a loadflow can be run before writing the diagram so that it displays active powers, for instance:

```
>>> network = pp.network.create_ieee9()
>>> result = pp.loadflow.run_ac(network)
>>> network.write_network_area_diagram_svg('ieee9.svg')
```

Network-area diagrams can be customized through NadParameters:

```
>>> from pypowsybl.network import NadParameters
>>> network = pp.network.create_ieee14()
>>> edge_info_parameters = EdgeInfoParameters(info_side_external=EdgeInfoType.ACTIVE_
↪POWER, info_middle_side1=EdgeInfoType.EMPTY, info_middle_side2=EdgeInfoType.EMPTY,␣
↪info_side_internal=EdgeInfoType.EMPTY)
>>> nad = network.get_network_area_diagram('VL6', nad_parameters=NadParameters(edge_name_
↪displayed=True, id_displayed=True, edge_info_along_edge=False, power_value_precision=1,
↪ angle_value_precision=0, current_value_precision=1, voltage_value_precision=0, bus_
↪legend=False, substation_description_displayed=True, edge_info_displayed=EdgeInfoType.
↪REACTIVE_POWER, voltage_level_details=False, injections_added=True, edge_info_
↪parameters=edge_info_parameters, scale_factor=1.0, timeout_seconds = 10, edge_info_
↪included=True, voltage_level_legends_included=True))

- edge_name_displayed (deprecated, use edge_info_parameters instead): if true, names␣
↪along lines and transformer legs are displayed (default value false)
- id_displayed: if true, the equipment ids are displayed. If false, the equipment names␣
↪are displayed (if a name is null, then the id is displayed) (default value false)
- edge_info_along_edge: if true, the edge information (P or Q values for example) is␣
↪displayed alongside the edge. If false, the edge information is displayed␣
↪perpendicularly to the edge (default value true)
- power_value_precision: number of digits after the decimal point for power values␣
↪(default value 0)
- angle_value_precision: number of digits after the decimal point for angle values␣
↪(default value 1)
- current_value_precision: number of digits after the decimal point for current values␣
↪(default value 0)
- voltage_value_precision: number of digits after the decimal point for voltage values␣
↪(default value 1)
- bus_legend: if true, angle and voltage values associated to a voltage level are␣
↪displayed in a text box. If false, only the voltage level name is displayed (default␣
↪value true)
- substation_description_displayed: if true, the substation name is added to the voltage␣
```

(continues on next page)

(continued from previous page)

```
↪level info on the diagram (default value false)
- edge_info_displayed (deprecated, use edge_info_parameters instead): type of info␣
↪displayed (EdgeInfoType.ACTIVE_POWER(default),EdgeInfoType.REACTIVE_POWER or␣
↪EdgeInfoType.CURRENT)
- voltage_level_details: if true, additional information about voltage levels is␣
↪displayed in text boxes. The content of the additional information is determined by␣
↪the label provider that is used.
- injections_added: if true, the injections present on the bus nodes of the voltage␣
↪levels are displayed.
- edge_info_parameters: type of info displayed (default value: info_side_
↪external=EdgeInfoType.ACTIVE_POWER, info_middle_side1=EdgeInfoType.EMPTY, info_middle_
↪side2=EdgeInfoType.EMPTY, info_side_internal=EdgeInfoType.EMPTY)
- scale_factor: factor of which the whole NAD will be scaled (default value 1.0)
- timeout_seconds: timeout in seconds for the force layout (default value 10.)
- edge_info_included: if true, the resulting SVG will contain the edge info data. If␣
↪false, this data will only be in the metadata.
- voltage_level_legends_included: if true, the resulting SVG will contain the voltage␣
↪level legends data. If false, this data will only be in the metadata.
```

In order to get a list of the displayed voltage levels from an input voltage level (or an input list of voltage levels) and a depth:

```
>>> network = pp.network.create_ieee300()
>>> list_vl = network.get_network_area_diagram_displayed_voltage_levels('VL1', 1)
```

We can generate a network area diagram using fixed positions, defined in a dataframe:

```
>>> import pandas as pd
>>> network = pp.network.create_ieee9()
>>> pos_df = pd.DataFrame.from_records(index='id',
                                       columns=['id', 'x', 'y',
                                                'legend_shift_x', 'legend_shift_y',
↪'legend_connection_shift_x', 'legend_connection_shift_y'],
                                       data=[
                                           ('VL5',  10.0, 20.0, 80.0, -30, 80.0, 0),
                                           ('VL6', 400.0, 20.0, 80.0, -30, 80.0, 0),
                                           ('VL8', 800.0, 20.0, 80.0, -30, 80.0, 0)
                                       ])
>>> nad = network.get_network_area_diagram(fixed_positions=pos_df)
```

In the dataframe:

- id is the equipment id for the node
- x, y define the position for the node
- legend_shift_x, legend_shift_y define the legend box top-left position (relative to the node position)
- legend_connection_shift_x, legend_connection_shift_y define the legend box side endpoint position (relative to the node position) for the segment connecting a node and its legend box

The optional parameter fixed_positions can also be set in the write_network_area_diagram function. Note that positions for elements not included in the dataframe are computed using the current layout algorithm.

We can further customize the NAD diagram using the NadProfile. For example, to set

- the labels for the branches, and the arrows direction

- the VL and BUS descriptions in the VL info boxes

by using dataframes:

```
>>> import pandas as pd
>>> network = pp.network.create_four_substations_node_breaker_network()
>>> labels_df = pd.DataFrame.from_records(index='id', columns=['id', 'side1', 'middle',
↪'side2', 'arrow1', 'arrow2'],
                                          data=[
                                              ('LINE_S2S3', 'L1_1', 'L1', 'L1_2', 'IN',
↪'IN'),
                                              ('LINE_S3S4', 'L2_1', 'L2', 'L2_2', 'OUT',
↪'IN'),
                                              ('TWT', 'TWT1_1', 'TWT1', 'TWT1_2', None,
↪'OUT')
                                          ])
>>> vl_descriptions_df=pd.DataFrame.from_records(index='id',
                                          data=[
                                             {'id': 'S1VL1', 'type': 'HEADER',
↪'description': 'VL A'},
                                             {'id': 'S1VL1', 'type': 'FOOTER',
↪'description': 'VL A footer'},
                                             {'id': 'S1VL2', 'type': 'HEADER',
↪'description': 'VL B'},
                                             {'id': 'S2VL1', 'type': 'HEADER',
↪'description': 'VL C'},
                                             {'id': 'S3VL1', 'type': 'HEADER',
↪'description': 'VL D'},
                                             {'id': 'S3VL1', 'type': 'FOOTER',
↪'description': 'VL D footer'}
                                          ])
>>> bus_descriptions_df=pd.DataFrame.from_records(index='id',
                                          data=[
                                              {'id': 'S1VL1_0', 'description': 'BUS A'},
                                              {'id': 'S1VL2_0', 'description': 'BUS B'},
                                              {'id': 'S2VL1_0', 'description': 'BUS C'},
                                              {'id': 'S3VL1_0', 'description': 'BUS D'}
                                          ])
>>> bus_node_style_df = pd.DataFrame.from_records(index='id',
                                          data=[
                                              {'id': 'S1VL1_0', 'fill': 'red', 'edge':
↪'black', 'edge-width': '4px'},
                                              {'id': 'S1VL2_0', 'fill': 'blue', 'edge':
↪'black', 'edge-width': '4px'},
                                              {'id': 'S2VL1_0', 'fill': 'yellow', 'edge
↪': 'black', 'edge-width': '4px'},
                                          ])
>>> edge_style_df = pd.DataFrame.from_records(index='id',
                                       data=[
                                              {'id': 'LINE_S2S3', 'edge1': 'blue', 'width1':
↪'16px', 'dash1': '12,12' ,'edge2': 'blue', 'width2': '16px', 'dash2': '12,12'},
                                          {'id': 'LINE_S3S4', 'edge1': 'green',  'width1
↪': '3px', 'edge2': 'green',  'width2': '3px'},
                                          {'id': 'TWT'      , 'edge1': 'yellow', 'width1
```

(continues on next page)

(continued from previous page)

```
↪': '4px', 'edge2': 'blue',  'width2': '4px'},
                                        ])
>>> diagram_profile=pp.network.NadProfile(branch_labels=labels_df, vl_descriptions=vl_
↪descriptions_df, bus_descriptions=bus_descriptions_df,
                                    bus_node_styles=bus_node_style_df, edge_styles=edge_
↪style_df)
>>> pars=pp.network.NadParameters(edge_name_displayed=True)
>>> network.get_network_area_diagram(voltage_level_ids='S1VL1', depth=2, nad_
↪parameters=pars, nad_profile=diagram_profile)
```

In the branch_labels dataframe parameter:

- id is the branch id
- side1 and side2 define the labels along the two branch's edges
- middle defines the branch's label
- arrow1 and arrow2 define the direction of the arrows at the ends of the branch: 'IN' or 'OUT'. None (or an empty string) does not display the arrow.

In the vl_descriptions dataframe parameter:

- id is the VL id
- type: 'HEADER' or 'FOOTER' determines if the descrtiption appears above or below the bus description, in the VL info box
- description define a label for the VL. Entries with the same VL id are displayed sequentially as multiple rows

In the bus_descriptions dataframe parameter:

- id is the BUS id
- description define a label for the BUS

In the bus_node_styles dataframe parameter:

- id is the BUS id
- fill is the fill color for the node
- edge is the edge color for the node
- width is the width of the edge for the node

In the edge_styles dataframe parameter:

- id is the branch id
- edge1, width1 and dash1 is the color, the width and the dash pattern for the first branch's edge
- edge2, width2 and dash2 is the color, the width and the dash pattern for the second branch's edge

The dash pattern string specifies the lengths of alternating dashes and gaps in the edge, separated by commas and/or spaces

An additional three_wt_labels dataframe parameter can be used to set the labels and the arrows direction for three winding transformers:

- id is the three winding transformer id
- side1, side2, and side3 define the labels along the three winding transformer legs

- arrow1, arrow2, and arrow3 define the direction of the arrows at the ends of the three winding transformer legs: 'IN' or 'OUT'. None (or an empty string) does not display the arrow.

Similarly to the edge_styles, the three_wt_styles parameter can be used to set the style for the three winding transformers:

- id is the three winding transformer id
- edge1, width1 and dash1 is the color, the width and the dash pattern for the first transformer's leg
- edge2, width2 and dash2 is the color, the width and the dash pattern for the second transformer's leg
- edge3, width3 and dash3 is the color, the width and the dash pattern for the third transformer's leg

The optional parameter nad_profile can also be set in the write_network_area_diagram function.

We can create a NadProfile, initialized with default content (in terms of branch_labels, vl_descriptions, and bus_descriptions), by using the get_default_nad_profile function:

```
>>> default_profile = network.get_default_nad_profile()
```

This is useful for example if we want to update just a few labels and rely on the defaults for the other ones.

Network area diagram using geographical data

We can load a network with geographical data (in WGS84 coordinates system) for substations and lines (in that case, the geographical positions represent the line path). One way to do that is to load a CGMES file containing a GL profile (Graphical Layout). By default this profile is not read. To activate GL profile loading and creation of substations ans lines geographical positions in the PowSyBl network model we have to pass an additional parameter to the load function.

```
>>> network = pp.network.load('MicroGridTestConfiguration_T4_BE_BB_Complete_v2.zip', {
↪'iidm.import.cgmes.post-processors': 'cgmesGLImport'})
```

We can now check loaded position by displaying *SubstationPosition* and *LinePosition* extensions.

```
>>> network.get_extension('substationPosition')
                                  latitude  longitude
id
87f7002b-056f-4a6a-a872-1744eea757e3   51.3251    4.25926
37e14a0f-5e34-4647-a062-8bfd9305fa9d   50.8038    4.30089
```

```
>>> network.get_extension('linePosition')
                                      latitude  longitude
id                                   num
b58bf21a-096a-4dae-9a01-3f03b60c24c7 0     50.8035    4.30113
                                     1     50.9169    4.34509
                                     2     51.0448    4.29565
                                     3     51.1570    4.38354
                                     4     51.3251    4.25926
ffbabc27-1ccd-4fdc-b037-e341706c8d29 0     50.8035    4.30113
                                     1     50.9169    4.34509
                                     2     51.0448    4.29565
                                     3     51.1570    4.38354
                                     4     51.3251    4.25926
```

When we generate a network area diagram, an automatic force layout is performed by default. The diagram looks like this:

```
>>> network.write_network_area_diagram('be.svg')
```

Now that we have geographical positions in our data model, we can change the layout to render the diagram with the geographical layout:

```
>>> parameter = pp.network.NadParameters(layout_type=pp.network.NadLayoutType.
↪GEOGRAPHICAL)
>>> network.write_network_area_diagram('be.svg', nad_parameters=parameter)
```

Display diagrams using Jupyter widgets

You can also display diagrams through Jupyter widgets.

Get a handle on Jupyter widgets

You can use the explorer to check the example notebooks given in the pypowsybl-jupyter repository:

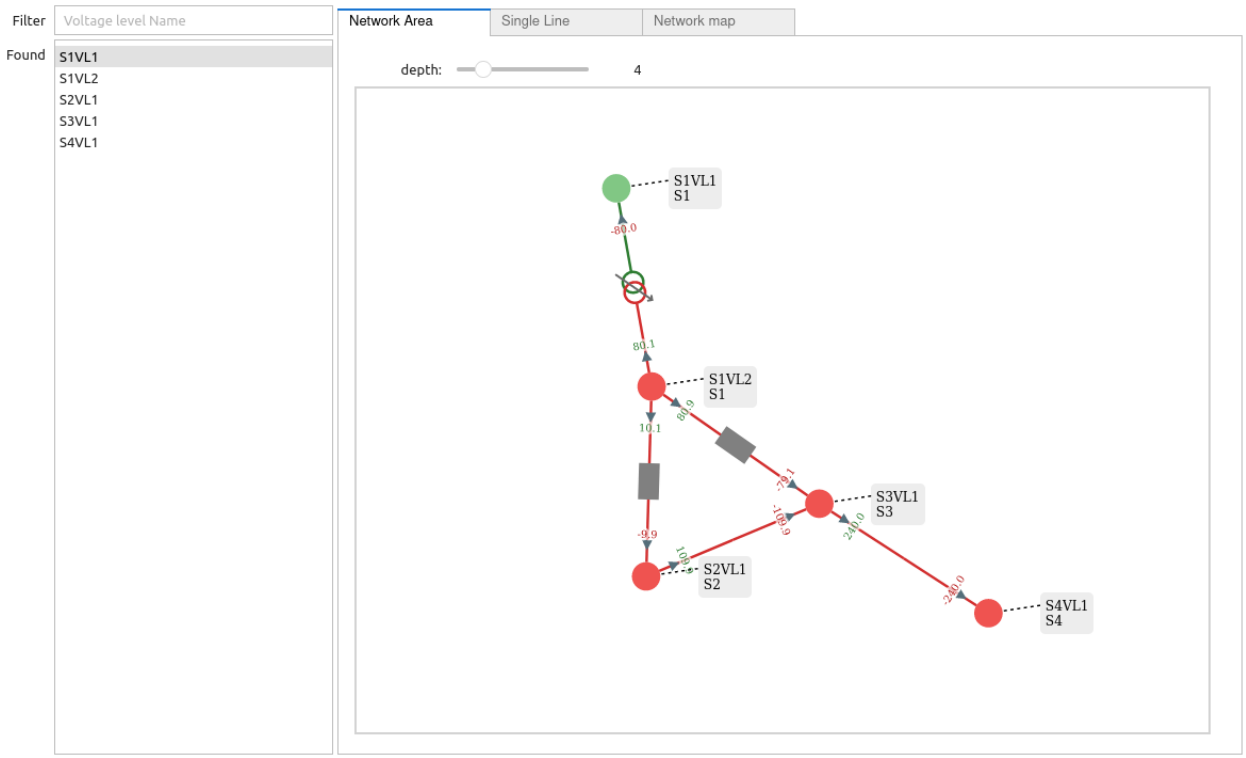
```
pip install pypowsybl_jupyter
jupyter lab
```

The network_explorer features three tabs:

- A tab for network-area diagrams;
- A tab for single-line diagrams;
- A tab for a map viewer.

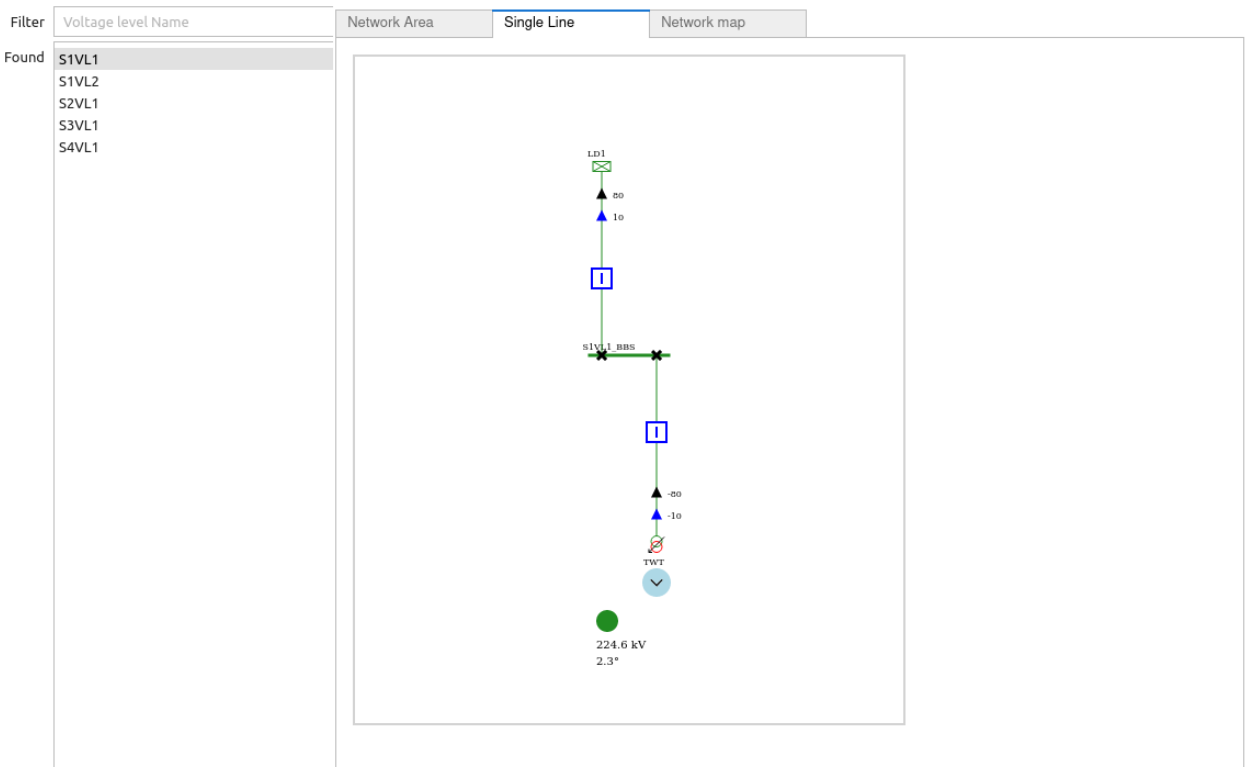
Explore the network_explorer tabs

- **The network-area diagram tab**



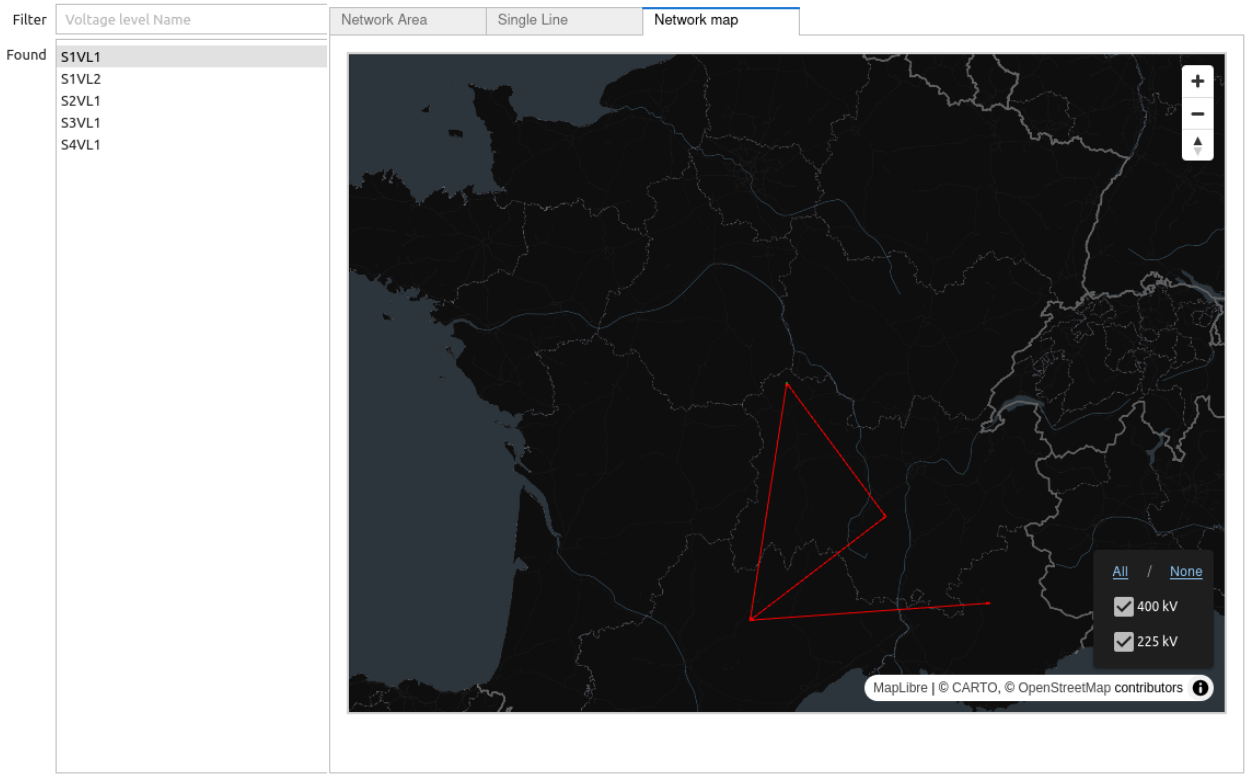
The network-area diagram tab displays the network-area diagram of the selected voltage level in the left column at the desired depth. The depth is user-defined, thanks to the slide button above the display zone.

- **The single-line diagram tab**

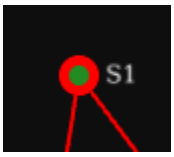


The single-line diagram tab displays the single-line diagram of the selected voltage level in the left column. You can navigate from voltage level to voltage level using the circled arrows.

- **The map viewer tab**



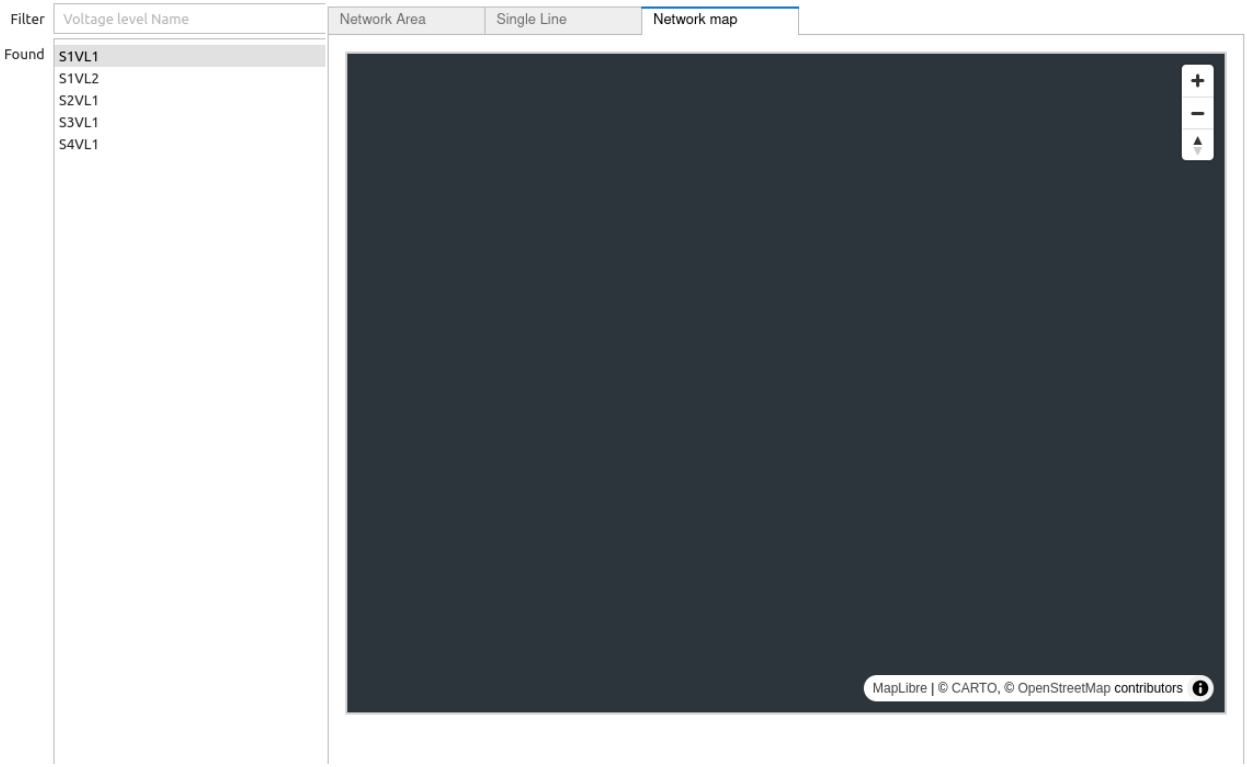
The map viewer tab displays a geographical representation of the network with a background map. The vertices of the graph are substations and the edges are lines, tie lines or HVDC lines. Voltage levels are represented as concentric circles inside a substation:



Selecting a voltage level on the left column will center the map on the corresponding substation.

Users can filter the displayed voltage levels through a nominal voltage filter. By default, only higher nominal voltages are selected.

Please note that if no geographical extensions are available for substations, the tab will be blank.



Go further

Check the complete documentation available on the widgets here.

2.1.3 Per Unit data

PyPowSyBl provides methods to per unit the scientific data. They are part of the network api. To per-unit the data, the attribute per_unit of the network has to be set.

```
>>> net = pp.network.create_four_substations_node_breaker_network()
>>> net.per_unit=True
>>> net.get_lines()
          name         r         x   g1   b1   g2   b2        p1        q1        i1    ␣
↪   p2        q2        i2 voltage_level1_id voltage_level2_id  bus1_id  bus2_id ␣
↪connected1  connected2
id
LINE_S2S3       0.000006  0.011938  0.0  0.0  0.0  0.0  1.098893  1.900229  2.147594 -1.
↪098864 -1.845171  2.147594             S2VL1             S3VL1  S2VL1_0  S3VL1_0      ␣
↪ True        True
LINE_S3S4       0.000006  0.008188  0.0  0.0  0.0  0.0  2.400036  0.021751  2.400135 -2.
↪400000  0.025415  2.400135             S3VL1             S4VL1  S3VL1_0  S4VL1_0      ␣
↪ True        True
```

For example for lines r, x, g1, b1, g2, b2, p1, q1, i1, p2, q2, i2 are per united according to the nominal voltage and to the nominal apparent power. The nominal apparent power is by default 100 MVA. It can be set like this :

```
>>> net.nominal_apparent_power=250
>>> net.get_lines()
          name         r         x   g1   b1   g2   b2        p1        q1        i1    ␣
```

(continues on next page)

(continued from previous page)

```
↪   p2        q2        i2 voltage_level1_id voltage_level2_id  bus1_id  bus2_id ␣
↪connected1  connected2
id
LINE_S2S3       0.000016  0.029844  0.0  0.0  0.0  0.0  0.439557  0.760092  0.859038 -0.
↪439546 -0.738068  0.859037             S2VL1             S3VL1  S2VL1_0  S3VL1_0      ␣
↪ True        True
LINE_S3S4       0.000016  0.020469  0.0  0.0  0.0  0.0  0.960014  0.008700  0.960054 -0.
↪960000  0.010166  0.960054             S3VL1             S4VL1  S3VL1_0  S4VL1_0      ␣
↪ True        True
```

2.1.4 Per Unit formula

Resistance R

for network elements with only one nominal voltage :

$$\frac{S_n}{V_{nominal}^2} R$$

with Sn the nominal apparent power For two winding transformers, the nominal voltage is the nominal voltage of the side 2 For lines, it is according to both sides :

$$\frac{S_n}{V_{nominal1} V_{nominal2}} R$$

Reactance X

for network elements with only one nominal voltage :

$$\frac{S_n}{V_{nominal}^2} X$$

with Sn the nominal apparent power For two winding transformers, the nominal voltage is the nominal voltage of the side 2 For lines, it is according to both sides :

$$\frac{S_n}{V_{nominal1} V_{nominal2}} X$$

Susceptance B

for network elements with only one nominal voltage :

$$\frac{V_{nominal}^2}{S_n} B$$

with Sn the nominal apparent power For two winding transformers, to compute B, the nominal voltage is the nominal voltage of the side 2 For lines, B is **on side one** according to both sides :

$$\frac{V_{nom1}^2 B + (V_{nominal1} - V_{nominal2}) V_{nominal1} Im(Y)}{S_n}$$

where Y is the admittance (Y = 1/Z where Z is the impedance) and Im() the imaginary part

Conductance G

for network elements with only one nominal voltage :

$$\frac{V_{nominal}^2}{S_n}G$$

with Sn the nominal apparent power For two winding transformers, to compute G, the nominal voltage is the nominal voltage of the side 2 For lines, G is **on side one** according to both sides :

$$\frac{V_{nom1}^2 G + (V_{nominal1} - V_{nominal2})V_{nominal1}Re(Y)}{S_n}$$

where Y is the admittance (Y = 1/Z where Z is the impedance) and Re() the real part for side 2 just inverse Vnominal1 and Vnominal2

Voltage V

$$\frac{V}{V_{nominal}}$$

the voltage is perunit by the nominal voltage. For network element with a target voltage, it per united by the nominal voltage of the target element.

Active Power P

$$\frac{P}{S_n}$$

with Sn the nominal apparent power

Reactive Power Q

$$\frac{Q}{S_n}$$

with Sn the nominal apparent power

Electric Current I

$$\frac{\sqrt{3}V_{nominal}}{S_n 10^3}I$$

with Sn the nominal apparent power

Angle

the angle are in degrees in PyPowSyBl, but when per-unit is activated it is in radian even if it is not really related to per-uniting.

2.1.5 Running a load flow

You can use the module *pypowsybl.loadflow* in order to run load flows on networks.

Start by importing the module:

```
import pypowsybl.network as pn
import pypowsybl.loadflow as lf
```

Providers

We can get the list of supported load flow implementations (so called providers) and default one:

```
>>> lf.get_provider_names()
['DynaFlow', 'OpenLoadFlow']
>>> lf.get_default_provider()
'OpenLoadFlow'
```

By default, load flows are based on the OpenLoadFlow implementation, fully described here. OpenLoadFlow supports AC Newton-Raphson and linear DC calculation methods.

You may also use DynaFlow, provided by the Dynawo project. DynaFlow is a new steady-state simulation tool that aims at calculating the steady-state point by using a simplified time-domain simulation. Please see configuration instructions here.

Parameters

The most important part before running a load flow is knowing the parameters and change them if needed. Let's have a look at the default ones:

```
>>> lf.Parameters()
Parameters(voltage_init_mode=UNIFORM_VALUES, transformer_voltage_control_on=False, use_
↪reactive_limits=True, phase_shifter_regulation_on=False, twt_split_shunt_
↪admittance=False, shunt_compensator_voltage_control_on=False, read_slack_bus=True,␣
↪write_slack_bus=True, distributed_slack=True, balance_type=PROPORTIONAL_TO_GENERATION_
↪P_MAX, dc_use_transformer_ratio=True, countries_to_balance=[], component_mode=
↪<ComponentMode.MAIN_CONNECTED: 0>, hvdc_ac_emulation=True, dc_power_factor=1.0,␣
↪dc=False, provider_parameters={})
```

For more details on each parameter, please refer to the *API reference*.

All parameters are also fully described in Powsybl load flow parameters documentation.

Parameters specific to a provider

Some parameters are not supported by all load flow providers but specific to only one. These specific parameters could be specified in a less typed way than common parameters using the *provider_parameters* attribute.

Warning

provider_parameters is a dictionary in which all keys and values **must** be a string, even in case of a numeric value:

- string and integer parameters do not bring much challenge:

  ```
  provider_parameters={'someStringParam' :  'myStringValue', 'someIntegerParam' :
  '42'}
  ```

- for float (double) parameters, use the dot as decimal separator. E notation is also supported:

  ```
  provider_parameters={'someDoubleParam' :  '1.23', 'someOtherDoubleParam' :  '4.
  56E-2'}
  ```

- for boolean parameters, use either *'True'*, *'true'*, *'False'*, *'false'*:

  ```
  provider_parameters={'someBooleanParam' :  'true'}
  ```

- for string list parameters, use the comma as a separator:

  ```
  provider_parameters={'someStringListParam' :  'value1,value2,value3'}
  ```

We can list supported parameters specific to default provider using:

```
>>> lf.get_provider_parameters_names()
['slackBusSelectionMode', 'slackBusesIds', 'lowImpedanceBranchMode',
↪'voltageRemoteControl', ...]
```

And get more detailed information about theses parameters, such as parameter description, type, default value if any, possible values if applicable, using:

```
>>> lf.get_provider_parameters().query('name == "slackBusSelectionMode" or name ==
↪"slackBusesIds"')
                          category_key               description         type␣
↪default                                possible_values
name
slackBusSelectionMode  SlackDistribution  Slack bus selection mode       STRING  MOST_
↪MESHED  [FIRST, MOST_MESHED, NAME, LARGEST_GENERATOR]
slackBusesIds          SlackDistribution             Slack bus IDs  STRING_LIST
```

For instance, OLF supports configuration of slack bus from its ID like this:

```
>>> p = lf.Parameters(provider_parameters={'slackBusSelectionMode' : 'NAME',
↪'slackBusesIds' : 'VLHV2_0'})
```

AC Load Flow

In order to run an AC loadflow, simply use the *run_ac()* method:

```
>>> network = pn.create_eurostag_tutorial_example1_network()
>>> results = lf.run_ac(network, parameters=lf.Parameters(distributed_slack=False))
```

The result is composed of a list of component results, one for each connected component of the network included in the computation:

```
>>> results
[ComponentResult(connected_component_num=0, synchronous_component_num=0,␣
↪status=CONVERGED, status_text=Converged, iteration_count=3, reference_bus_id='VLHV1_0',
↪ slack_bus_results=[SlackBusResult(id='VLHV1_0', active_power_mismatch=-606.5596...)],␣
↪distributed_active_power=0.0)]
```

Component results provides general information about the loadflow execution: was it successful? How many iterations did it need? What is the remaining active power imbalance? For example, let's have a look at the imbalance on the main component of the network:

```
>>> results[0].slack_bus_results[0].active_power_mismatch
-606.5596...
```

Then, the main output of the loadflow is actually the updated data in the network itself: all voltages and flows are now updated with the computed values. For example you can have a look at the voltage magnitudes (rounded to 2 digits here):

```
>>> network.get_buses().v_mag.round(2)
id
```

(continues on next page)

(continued from previous page)

```
VLGEN_0      24.50
VLHV1_0     400.62
VLHV2_0     388.33
VLLOAD_0    146.90
Name: v_mag, dtype: float64
```

DC Load Flow

In order to run a DC loadflow, simply use the *run_dc()* method.

For that example, we will use a distributed slack, with imbalance distributed on generators, proportional to their maximum power. We also choose to ignore transformer ratios in the DC equations:

```
>>> parameters = lf.Parameters(dc_use_transformer_ratio=False, distributed_slack=True,
...                            balance_type=lf.BalanceType.PROPORTIONAL_TO_GENERATION_P_
↪MAX)
```

Then let's create our test network and run the DC load flow:

```
>>> network = pn.create_eurostag_tutorial_example1_network()
>>> results = lf.run_dc(network, parameters)
```

We can finally retrieve the computed flows on lines:

```
>>> network.get_lines()[['p1', 'p2']]
                p1     p2
id
NHV1_NHV2_1  300.0 -300.0
NHV1_NHV2_2  300.0 -300.0
```

Reports

Reports contain detailed computation information. To see those reports, pass a report_node argument to the run command.

```
>>> report_node = pp.report.ReportNode()
>>> network = pn.create_eurostag_tutorial_example1_network()
>>> results = lf.run_ac(network, parameters, report_node=report_node)
>>> print(report_node)
+
   + Load flow on network 'sim1'
      + Network CC0 SC0
         + Network info
            Network has 4 buses and 4 branches
            Network balance: active generation=1214.0 MW, active load=600.0 MW, reactive␣
↪generation=0.0 MVar, reactive load=200.0 MVar
            Angle reference bus: VLHV1_0
            Slack bus: VLHV1_0
         + Outer loop DistributedSlack
            + Outer loop iteration 1
               Slack bus active power (-606.5596837558763 MW) distributed in 1␣
↪distribution iteration(s)
            + Outer loop iteration 2
```

(continues on next page)

(continued from previous page)

```
              Slack bus active power (-1.8792855272990572 MW) distributed in 1␣
↪distribution iteration(s)
        Outer loop VoltageMonitoring
        Outer loop ReactiveLimits
        Outer loop DistributedSlack
        Outer loop VoltageMonitoring
        Outer loop ReactiveLimits
        AC load flow completed successfully (solverStatus=CONVERGED,␣
↪outerloopStatus=STABLE)
```

Asynchronous API

An asynchronous API based on Python's asyncio has been added for AC loadflow calculations (DC loadflow support will be added in a future release). The following example demonstrates how to:

- Load a network
- Create two identical variants from the initial state
- Run AC loadflow calculations on both variants concurrently
- Wait for both results and display their convergence status

Both loadflow calculations are executed in parallel using Python's asyncio API.

Caution

The network has to be loaded using a special parameter *allow_variant_multi_thread_access* to *True* to be able to work on multiple variants of a same network concurrently using different threads.

```
>>> import asyncio
>>> async def run_2_lf():
...     lf1 = lf.run_ac_async(network, "variant1")
...     lf2 = lf.run_ac_async(network, "variant2")
...     results = await asyncio.gather(lf1, lf2)
...     print(results[0][0].status)
...     print(results[1][0].status)
>>> network = pn.create_ieee14(allow_variant_multi_thread_access=True)
>>> network.clone_variant("InitialState", "variant1")
>>> network.clone_variant("InitialState", "variant2")
>>> asyncio.run(run_2_lf())
ComponentStatus.CONVERGED
ComponentStatus.CONVERGED
```

2.1.6 Running a RAO

The RAO is currently in **beta** version.

You can use the module *pypowsybl.rao* in order to perform a remedial actions optimization on a network.

For detailed documentation of involved classes and methods, please refer to the *API reference*.

For detailed documentation of the Powsybl OpenRAO please refer to the PowSyBl RAO documentation.

Inputs for a RAO

To run a RAO you need:

- a network in a PyPowsybl supported exchange format
- a CRAC file (Contingency list, Remedial Actions and additional Constraints) in json
- optionally a GLSK file (Generation and Load Shift Keys) in json
- optionally a parameters file, in json, allowing to override the RAO parameters

Here is a code example of how to configure and run the RAO:

```
>>> import pypowsybl as pp
>>> from pypowsybl.rao import Parameters as RaoParameters
>>> from pypowsybl.rao import Crac
>>> from pypowsybl.rao import Glsk as RaoGlsk
>>> from pypowsybl.rao import RaoResult
>>>
>>> network =  pp.network.load(str(DATA_DIR.joinpath("rao/rao_network.uct")))
>>> parameters = RaoParameters.from_file_source(str(DATA_DIR.joinpath("rao/rao_
↪parameters.json")))
>>> rao_runner = pp.rao.create_rao()
>>> crac = Crac.from_file_source(network, str(DATA_DIR.joinpath("rao/rao_crac.json")))
>>> glsk = RaoGlsk.from_file_source(str(DATA_DIR.joinpath("rao/rao_glsk.xml")))
>>> rao_result = rao_runner.run(crac=crac, network=network, parameters=parameters, loop_
↪flow_glsk=glsk)
>>> rao_result.status()
<RaoComputationStatus.DEFAULT: 0>
```

Monitoring API

Rao monitoring can run through the following API using a rao result already produced by a run, or loaded from file :

```
>>> result_with_voltage_monitoring = rao_runner.run_voltage_monitoring(crac, network,␣
↪rao_result)
>>> monitoring_glsk = RaoGlsk.from_file_source(str(DATA_DIR.joinpath("rao/GlskB45test.xml
↪")))
>>> result_with_angle_monitoring = rao_runner.run_angle_monitoring(crac, network, rao_
↪result, monitoring_glsk=monitoring_glsk)
```

The returned rao result object are the original result enhanced with voltage or angle monitoring data.

Outputs of a RAO

The RAO results can be explored through the *RaoResult* object returned by the run function of the rao runner. Results are exposed in pandas dataframe format using the following API.

Retrieve the global result status (can be DEFAULT, FAILURE or PARTIAL_FAILURE):

```
>>> rao_result.status()
<RaoComputationStatus.DEFAULT: 0>
```

Retrieve the result for the flow CNEC:

```
>>> flow_cnec = rao_result.get_flow_cnec_results()
>>> flow_cnec.columns
Index(['cnec_id', 'optimized_instant', 'contingency', 'side', 'flow', 'margin',
       'relative_margin', 'commercial_flow', 'loop_flow', 'ptdf_zonal_sum'],
      dtype='object')
```

Each line represent a flow cnec result for an optimized instant and a contingency context.

When monitoring has been executed, voltage and angle cnec results can also be retrieved through pandas dataframes:

```
>>> voltage_cnec = rao_result.get_voltage_cnec_results()
>>> voltage_cnec.columns
Index(['cnec_id', 'optimized_instant', 'contingency', 'side', 'min_voltage',
       'max_voltage', 'margin'],
      dtype='object')
>>> angle_cnecs = rao_result.get_angle_cnec_results()
>>> angle_cnecs.columns
Index(['cnec_id', 'optimized_instant', 'contingency', 'angle', 'margin'], dtype='object')
```

Remedial action results are also available in a pandas dataframe :

```
>>> ra_results = rao_result.get_remedial_action_results()
>>> ra_results.columns
Index(['remedial_action_id', 'optimized_instant', 'contingency'], dtype='object')
```

For each remedial action, optimized instant and a contingency (if applicable) the activation information is available. For range actions the optimized tap is also available for PstRangeAction and optimized set point for all other RangeActions. Optimized tap and optimized set point are set to NaN when not applicable (not a range action).

It is possible to get the results of activated remedial actions for a specific type of remedial action only.

For network actions:

```
>>> ra_results = rao_result.get_network_action_results()
>>> ra_results.columns
Index(['remedial_action_id', 'optimized_instant', 'contingency'], dtype='object')
```

For PST range actions:

```
>>> ra_results = rao_result.get_pst_range_action_results()
>>> ra_results.columns
Index(['remedial_action_id', 'optimized_instant', 'contingency',
       'optimized_tap'],
      dtype='object')
```

For other non-PST range actions:

```
>>> ra_results = rao_result.get_range_action_results()
>>> ra_results.columns
Index(['remedial_action_id', 'optimized_instant', 'contingency',
       'optimized_set_point'],
      dtype='object')
```

Finally cost results can also be retrieved. Generic cost results are available in a dataframe :

```
>>> cost_results = rao_result.get_cost_results()
>>> cost_results.columns
Index(['functional_cost', 'virtual_cost', 'cost'], dtype='object')
```

With optimized instant as an index, functional cost, virtual cost and the sum of the two as cost for each optimized instant are available. Details for virtual cost can also be queried for a given virtual cost with the list of virtual cost names available. Cost for a given virtual cost name is returned as a pandas dataframe with cost value for each instant.

```
>>> virtual_cost_names = rao_result.get_virtual_cost_names()
>>> virtual_cost_names
['mnec-cost', 'sensitivity-failure-cost', 'loop-flow-cost']
>>> sensi_cost = rao_result.get_virtual_cost_results('sensitivity-failure-cost')
>>> sensi_cost.index
Index(['initial', 'preventive', 'outage', 'auto', 'curative'], dtype='object', name=
↪'optimized_instant')
>>> sensi_cost.columns
Index(['sensitivity-failure-cost'], dtype='object')
>>> sensi_cost.loc['curative', 'sensitivity-failure-cost']
np.float64(0.0)
```

The 'RaoResult' object can also be serialized to json and loaded from file:

```
>>> rao_result.serialize(str(DATA_DIR.joinpath("rao/results.json")))
>>> loaded_result = RaoResult.from_file_source(crac, str(DATA_DIR.joinpath("rao/results.
↪json")))
```

Rao logs filter

Open rao logs can be retrieved in the global powsybl logger. However if a user is only interested in the logs coming from open rao, a RaoLogFilter is available :

```
>>> import pypowsybl as pp
>>> import logging
>>> import sys
>>> from pypowsybl.rao import (Parameters as RaoParameters, RaoLogFilter, Crac, Glsk as␣
↪RaoGlsk)
>>>
>>> network =  pp.network.load(str(DATA_DIR.joinpath("rao/rao_network.uct")))
>>> parameters = RaoParameters.from_file_source(str(DATA_DIR.joinpath("rao/rao_
↪parameters.json")))
>>> rao_runner = pp.rao.create_rao()
>>> crac = Crac.from_file_source(network, str(DATA_DIR.joinpath("rao/rao_crac.json")))
>>> glsk = RaoGlsk.from_file_source(str(DATA_DIR.joinpath("rao/rao_glsk.xml")))
>>>
>>> logging.basicConfig(stream=sys.stdout) # Setup logging
>>> logger = logging.getLogger('powsybl')
>>> logger.setLevel(logging.ERROR)
>>> logger.addFilter(RaoLogFilter())
>>> rao_result = rao_runner.run(crac, network, parameters, loop_flow_glsk=glsk)
```

2.1.7 Running a security analysis

You can use the module *pypowsybl.security* in order to perform a security analysis on a network. Please check out the examples below.

For detailed documentation of involved classes and methods, please refer to the *API reference*.

AC security analysis

To perform a security analysis, you need at least a network and a contingency on this network. In the result there are violations detected with the initial loadflow on the network. These violations are collected in pre_contingency_result. The results contain also the violations created by the contingency, they are collected by contingency in post_contingency_results:

```
>>> network = pp.network.create_eurostag_tutorial_example1_network()
>>> network.update_loads(id='LOAD', p0=800)
>>> security_analysis = pp.security.create_analysis()
>>> security_analysis.add_single_element_contingency('NHV1_NHV2_1', 'First contingency')
>>> result = security_analysis.run_ac(network)
>>> result.pre_contingency_result
PreContingencyResult(, status=CONVERGED, limit_violations=[3])
>>> result.post_contingency_results
{'First contingency': PostContingencyResult(contingency_id='First contingency',␣
↪status=CONVERGED, limit_violations=[3])}
>>> result.limit_violations
                           subject_name   limit_type limit_name   limit  acceptable_
↪duration  limit_reduction        value side
contingency_id    subject_id
                  NHV1_NHV2_1                  CURRENT  permanent   500.0          ␣
↪2147483647              1.0   623.568946  ONE
                  NHV1_NHV2_2                  CURRENT  permanent   500.0          ␣
↪2147483647              1.0   655.409876  TWO
                  VLHV1                    LOW_VOLTAGE              400.0          ␣
↪2147483647              1.0   398.917401
First contingency NHV1_NHV2_2                  CURRENT        20'  1200.0              ␣
↪    60              1.0  1438.021676  ONE
                  NHV1_NHV2_2                  CURRENT  permanent   500.0          ␣
↪2147483647              1.0  1477.824335  TWO
                  VLHV1                    LOW_VOLTAGE              400.0          ␣
↪2147483647              1.0   392.158685
```

It is also possible to get a JSON file with the full security analysis results, just by using the *export_to_json* method, like in the example below :

```
>>> n = pp.network.create_eurostag_tutorial_example1_network()
>>> sa = pp.security.create_analysis()
>>> sa_result = sa.run_ac(n)
>>> sa_result.export_to_json(str(DATA_DIR.joinpath('json_file_security_analysis.json')))
```

Adding monitored Elements

This feature is used to get information on different element of the network after the loadflow's computations. Information can be obtained on buses, branches and three windings transformers.

```
>>> network = pp.network.create_eurostag_tutorial_example1_with_more_generators_network()
>>> security_analysis = pp.security.create_analysis()
>>> security_analysis.add_single_element_contingency('NHV1_NHV2_1', 'NHV1_NHV2_1')
>>> security_analysis.add_single_element_contingency('GEN', 'GEN')
>>> security_analysis.add_monitored_elements(voltage_level_ids=['VLHV2'])
>>> security_analysis.add_postcontingency_monitored_elements(branch_ids=['NHV1_NHV2_2'],␣
↪contingency_ids=['NHV1_NHV2_1', 'GEN'])
>>> security_analysis.add_postcontingency_monitored_elements(branch_ids=['NHV1_NHV2_1'],␣
↪contingency_ids='GEN')
>>> security_analysis.add_precontingency_monitored_elements(branch_ids=['NHV1_NHV2_2'])
>>> results = security_analysis.run_ac(network)
>>> results.bus_results
                                                          v_mag  v_angle
contingency_id operator_strategy_id voltage_level_id bus_id
                                    VLHV2            NHV2   389.95    -3.51
>>> results.branch_results
                                                        p1     q1       i1      p2      q2  ␣
↪    i2  flow_transfer
contingency_id operator_strategy_id branch_id
                                    NHV1_NHV2_2 302.44  98.74   456.77 -300.43 -137.19  ␣
↪488.99            NaN
GEN                                 NHV1_NHV2_1 302.44  98.74   456.77 -300.43 -137.19  ␣
↪488.99            NaN
                                    NHV1_NHV2_2 302.44  98.74   456.77 -300.43 -137.19  ␣
↪488.99            NaN
NHV1_NHV2_1                         NHV1_NHV2_2 610.56 334.06 1,008.93 -601.00 -285.38 1,
↪047.83            NaN
```

It also possible to get flow transfer on monitored branches in case of N-1 branch contingencies:

```
>>> n = pp.network.create_eurostag_tutorial_example1_network()
>>> sa = pp.security.create_analysis()
>>> sa.add_single_element_contingencies(['NHV1_NHV2_1', 'NHV1_NHV2_2'])
>>> sa.add_monitored_elements(branch_ids=['NHV1_NHV2_1', 'NHV1_NHV2_2'])
>>> sa_result = sa.run_ac(n)
>>> sa_result.branch_results
                                                          p1          q1           i1   ␣
↪      p2          q2           i2  flow_transfer
contingency_id operator_strategy_id branch_id
                                    NHV1_NHV2_2  302.444049   98.740275   456.768978 -
↪300.433895 -137.188493   488.992798            NaN
                                    NHV1_NHV2_1  302.444049   98.740275   456.768978 -
↪300.433895 -137.188493   488.992798            NaN
NHV1_NHV2_2                         NHV1_NHV2_1  610.562154  334.056272  1008.928788 -
↪600.996156 -285.379147  1047.825769       1.018761
NHV1_NHV2_1                         NHV1_NHV2_2  610.562154  334.056272  1008.928788 -
↪600.996156 -285.379147  1047.825769       1.018761
```

Operator strategies and remedial actions

Pypowsybl security analysis support operator strategies and remedial actions definition.

You can define several types of actions by calling the add_XXX_action API. All actions need a unique id to be referenced later at the operator strategy creation stage.

The supported actions in PyPowsybl are listed here:

- *switch*, to open or close a switch
- *phase_tap_changer_position*, to change the tap position of a phase tap changer
- *ratio_tap_changer_position*, to change the tap position of a ratio tap changer
- *load_active_power*, to change the active power of a load
- *load_reactive_power*, to change the reactive power of a load
- *shunt_compensator_position*, to change the section of a shunt compensator
- *generator_active_power*, to modify the generator active power
- *terminals_connection*, to connect/disconnect one or multiple sides of a network element

The following example defines a switch closing action with id 'SwitchAction' on the switch with id 'S4VL1_BBS_LD6_DISCONNECTOR'.

```
>>> n = pp.network.create_four_substations_node_breaker_network()
>>> sa = pp.security.create_analysis()
>>> sa.add_switch_action(action_id='SwitchAction', switch_id='S4VL1_BBS_LD6_DISCONNECTOR
↪', open=False)
```

To enable the application of the action you need to define an operator strategy and add the action to it. An operator strategy is a set of actions to be applied after the simulation of a contingency. It is defined with an unique id, a reference to the id of the contingency, a list action ids and a condition. The following operator strategy define the application of the switch action 'SwitchAction' after 'Breaker contingency' with the 'True' condition (always applied):

```
>>> n = pp.network.create_four_substations_node_breaker_network()
>>> sa = pp.security.create_analysis()
>>> sa.add_single_element_contingency(element_id='S4VL1_BBS_LD6_DISCONNECTOR',␣
↪contingency_id='Breaker contingency')
>>> sa.add_switch_action(action_id='SwitchAction', switch_id='S4VL1_BBS_LD6_DISCONNECTOR
↪', open=False)
>>> sa.add_operator_strategy(operator_strategy_id='OperatorStrategy1', contingency_id=
↪'Breaker contingency', action_ids=['SwitchAction'], condition_type=pp.security.
↪ConditionType.TRUE_CONDITION)
>>> sa.add_monitored_elements(branch_ids=['LINE_S3S4'])
>>> sa_result = sa.run_ac(n)
>>> df = sa_result.branch_results
>>> #Get the detailed results post operator strategy
>>> df.loc['Breaker contingency', 'OperatorStrategy1', 'LINE_S3S4']['p1'].item()
240.00360040333226
```

Results for the post remedial action state are available in the branch results indexed with the operator strategy unique id.

Adding limit reductions to the analysis

Limit reductions can be added to a security analysis in order to detect limit violations on lower values of operational limits, using the *add_limit_reductions* method.

The following example reduces by a factor of 0.8 all limits on the network for the security analysis:

```
>>> n = pp.network.create_eurostag_tutorial_example1_network()
>>> sa = pp.security.create_analysis()
>>> sa.add_single_element_contingency('NHV1_NHV2_1', 'First contingency')
>>> sa.add_limit_reductions(limit_type='CURRENT', permanent=True, temporary=True,␣
↪value=0.8)
>>> sa_result = sa.run_ac(n)
>>> sa_result.limit_violations["limit_reduction"].unique()
[0.8]
```

Limit reductions can also be more selective. They can only be applied to certain network elements using the *country*, *min_voltage* and *max_voltage* parameters, or to certain temporary limits using the *min_temporary_duration* and *max_temporary_duration* parameters (if *temporary=True*). For example, the following method adds a limit reduction of 0.9 on all temporary current limits with minimal acceptable duration of 300s, on all network elements in France with nominal voltage between 90 and 225kV :

```
sa..add_limit_reductions(limit_type='CURRENT', permanent=False, temporary=True, value=0.
↪9, min_temporary_duration=300, country='FR', min_voltage=90, max_voltage=225)
```

Warning

If multiple reductions are applicable to the same limits (eg: 0.9 on all limits and 0.8 on all temporary limits), only the last one in addition order will be applied for the security analysis. If a dataframe was used to create the limit reductions, the order of the lines is the addition order.

Adding input data from JSON files

It is possible to add the input data of a security analysis using JSON files. The contingencies can be added this way, using the *add_contingencies_from_json_file* method.

An example of a valid JSON contingency file is the following :

```
{
  "type" : "default",
  "version" : "1.0",
  "name" : "list",
  "contingencies" : [ {
    "id" : "contingency",
    "elements" : [ {
      "id" : "NHV1_NHV2_1",
      "type" : "BRANCH"
    }, {
      "id" : "NHV1_NHV2_2",
      "type" : "BRANCH"
    } ]
  }, {
    "id" : "contingency2",
```

(continues on next page)

(continued from previous page)

```
    "elements" : [ {
      "id" : "GEN",
      "type" : "GENERATOR"
    } ]
  } ]
}
```

From now on, it is possible to add the remedial actions using JSON files too, using the *add_actions_from_json_file* method. The following example is a valid JSON file input for this method :

```
{
  "version" : "1.0",
  "actions" : [ {
    "type" : "SWITCH",
    "id" : "id1",
    "switchId" : "S1VL2_LCC1_BREAKER",
    "open" : true
  }, {
    "type" : "SWITCH",
    "id" : "id2",
    "switchId" : "S1VL2_BBS2_COUPLER_DISCONNECTOR",
    "open" : true
  }]
}
```

Additionally, you can add operator strategies from JSON data, using the *add_operator_strategies_from_json_file* method. The following example is a valid JSON file input for this method :

```
{
  "version" : "1.1",
  "operatorStrategies" : [ {
    "id" : "id1",
    "contingencyContextType" : "SPECIFIC",
    "contingencyId" : "contingency",
    "conditionalActions" : [ {
      "id" : "stage1",
      "condition" : {
        "type" : "TRUE_CONDITION"
      },
      "actionIds" : [ "id1", "id2" ]
    } ]
  }]
}
```

2.1.8 Sensitivity analysis

For detailed documentation of the Powsybl sensitivity analysis please refer to the PowSyBl OpenLoadFlow documentation.

You can use the module *pypowsybl.sensitivity* in order to perform sensitivity analysis on a network.

DC sensitivity analysis

To perform a sensitivity analysis, you first need to define "factors" you want to compute. What we call a factor is the dependency of a function, typically the active power flow on a branch, to a variable, typically the active power injection of a generator, a load or a phase shifter.

To make the definition of those factors easier, `pypowsybl` provides a method to define the branches for which the flow sensitivity should be computed, and for which injections or phase shifters. We obtain a matrix of sensitivities as a result:

```
>>> import pypowsybl as pp
>>> network = pp.network.create_eurostag_tutorial_example1_network()
>>> analysis = pp.sensitivity.create_dc_analysis()
>>> analysis.add_branch_flow_factor_matrix(branches_ids=['NHV1_NHV2_1', 'NHV1_NHV2_2'],␣
↪variables_ids=['LOAD'])
>>> result = analysis.run(network)
>>> result.get_reference_matrix()
                  NHV1_NHV2_1  NHV1_NHV2_2
reference_values        300.0        300.0
>>> result.get_sensitivity_matrix()
      NHV1_NHV2_1  NHV1_NHV2_2
LOAD         -0.5         -0.5
```

Several matrix of sensitivity factors can be specified, in that case you must name your matrix at creation and reuse this name to query you results:

```
>>> import pypowsybl as pp
>>> network = pp.network.create_eurostag_tutorial_example1_network()
>>> analysis = pp.sensitivity.create_dc_analysis()
>>> analysis.add_branch_flow_factor_matrix(branches_ids=['NHV1_NHV2_1', 'NHV1_NHV2_2'],␣
↪variables_ids=['LOAD'], matrix_id='m1')
>>> analysis.add_branch_flow_factor_matrix(branches_ids=['NHV1_NHV2_1'], variables_ids=[
↪'GEN'], matrix_id='m2')
>>> result = analysis.run(network)
>>> result.get_reference_matrix('m1')
                  NHV1_NHV2_1  NHV1_NHV2_2
reference_values        300.0        300.0
>>> result.get_sensitivity_matrix('m1')
      NHV1_NHV2_1  NHV1_NHV2_2
LOAD         -0.5         -0.5
>>> result.get_sensitivity_matrix('m2')
     NHV1_NHV2_1
GEN          0.0
```

Zone to slack sensitivity

We illustrate this feature with a simple network where we have 4 countries (FR, DE, NL and BE) and 3 buses per countries. A zone is a group of weighted injections. With this network, we can create zones based on countries. The country attribute is defined in the network at the substation level through attribute *Country*.

First, we create a zone containing all generators of DE network with a shift key equals to generators' active power targets. To compute the sensitivity of an injection increase from this zone to the slack bus, we first create load flow parameters in order to disabled slack distribution. Note that this example is based on sensitivity analysis with DC approximation. In the following lines, we ask for the DE zone sensitivity on the border line BBE2AA1 FFRAA1 1.

```
>>> n = pp.network.load('simple-eu.uct')
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE')
>>> params = pp.loadflow.Parameters(distributed_slack=False)
>>> sa = pp.sensitivity.create_dc_analysis()
>>> sa.set_zones([zone_de])
>>> sa.add_branch_flow_factor_matrix(['BBE2AA1  FFR3AA1  1'], ['DE'], 'm')
>>> results = sa.run(n, params)
>>> m = results.get_sensitivity_matrix('m')
         BBE2AA1  FFR3AA1  1
DE             -0.45182
```

1 MW increase on DE zone and 1 MW decrease on slack bus injection is responsible of a variation of -0.45182 MW on border line BBE2AA1 FFRAA1 1.

Zone to zone sensitivity

This feature is better known as Power Transfer Distribution Factor (PTDF).

In the following example, we compute the sensitivity of a active power transfer from FR zone to DE zone on the border line 'BBE2AA1 FFRAA1 1', through two zone to slack sensitivity calculations.

```
>>> zone_fr = pp.sensitivity.create_country_zone(n, 'FR')
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE')
>>> params = pp.loadflow.Parameters(distributed_slack=False)
>>> sa = pp.sensitivity.create_dc_analysis()
>>> sa.set_zones([zone_fr, zone_de])
>>> sa.add_branch_flow_factor_matrix(['BBE2AA1  FFR3AA1  1'], ['FR', 'DE'], 'm')
>>> results = sa.run(n, params)
>>> m = results.get_sensitivity_matrix('m')
         BBE2AA1  FFR3AA1  1
FR            -0.708461
DE            -0.451820
```

1 MW active power transfer from FR zone to DE zone will be responsible of a variation of -0.256641 MW (indeed -0.708461 MW - (-0.451820 MW)) on the border line BBE2AA1 FFRAA1 1.

Let's obtain that directly. In the following example, we create four zones based on countries FR, DE, BE and NL. After a sensitivity analysis where we should set the zones, we are able to ask for a FR zone to slack sensitivity, a FR to DE zone to zone sensitivity, a DE to FR zone to zone sensitivity and a NL zone to slack sensitivity, on the border lines 'BBE2AA1 FFR3AA1 1' and 'FFR2AA1 DDE3AA1 1'.

```
>>> zone_fr = pp.sensitivity.create_country_zone(n, 'FR')
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE')
>>> zone_be = pp.sensitivity.create_country_zone(n, 'BE')
>>> zone_nl = pp.sensitivity.create_country_zone(n, 'NL')
>>> params = pp.loadflow.Parameters(distributed_slack=False)
>>> sa = pp.sensitivity.create_dc_analysis()
>>> sa.set_zones([zone_fr, zone_de, zone_be, zone_nl])
>>> sa.add_branch_flow_factor_matrix(['BBE2AA1  FFR3AA1  1', 'FFR2AA1  DDE3AA1  1'], ['FR
↪', ('FR', 'DE'), ('DE', 'FR'), 'NL'], 'm')
>>> result = sa.run(n, params)
>>> m = result.get_sensitivity_matrix('m')
          BBE2AA1  FFR3AA1  1  FFR2AA1  DDE3AA1  1
FR                 -0.708461            0.291539
```

(continues on next page)

(continued from previous page)

```
FR -> DE            -0.256641             0.743359
DE -> FR             0.256641            -0.743359
NL                  -0.225206            -0.225206
```

Sensitivity to a X-Node

X-Nodes when imported from a UCTE or CGMES file are represented by a so called "dangling line" in the PowSyBl network model. The dangling line ID is taken from the line ID connecting the X-Node. So to calculate a X-Node sensitivity, we just have to use the dangling line ID as the injection in the zone definition.

```
>>> n = pp.network.load('simple-eu-xnode.uct')
>>> n.get_dangling_lines()
                    name    r     x    g    b   p0   q0   p   q   i voltage_level_id    ␣
↪bus_id  connected pairing_key ucte_xnode_code isCoupler status_XNode geographicalName
id
NNL2AA1         XXXXXX11    1   0.0 10.0  0.0  0.0  0.0 0.0 NaN NaN              NaN   ␣
↪NNL2AA1       True   NNL2AA1_0       NNL2AA1_0    True       XXXXXX11       EQUIVALENT

>>> zone_x = pp.sensitivity.create_empty_zone("X")
```

We can see that the dangling line 'NNL2AA1 XXXXXX11 1' correspond to the X-Node XXXXXX11 (see column pairing_key of dangling line data frame). To calculate to sensitivity of X-Node XXXXXX11 on tie line 'BBE2AA1 FFR3AA1 1':

```
>>> zone_x.add_injection('NNL2AA1  XXXXXX11 1')
>>> sa = pp.sensitivity.create_dc_analysis()
>>> sa.set_zones([zone_x])
>>> sa.add_branch_flow_factor_matrix(['BBE2AA1  FFR3AA1  1'], ['X'], 'm')
>>> result = sa.run(n)
>>> result.get_sensitivity_matrix('m')
   BBE2AA1  FFR3AA1  1
X            0.176618
```

Shift keys modification

When we create a zone from a country, the default behaviour is to use the generator active power target as weight. It means that:

```
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE')
```

is totally equivalent to:

```
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE', pp.sensitivity.ZoneKeyType.
↪GENERATOR_TARGET_P)
```

There are two additional modes, using generator maximum active power or load active power target, as illustrated in the following lines:

```
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE', pp.sensitivity.ZoneKeyType.
↪GENERATOR_MAX_P)
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE', pp.sensitivity.ZoneKeyType.
↪LOAD_P0)
```

You can display the keys with:

```
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE')
>>> zone_de.shift_keys_by_injections_ids
{'DDE1AA1 _generator': 2500.0,
 'DDE2AA1 _generator': 2000.0,
 'DDE3AA1 _generator': 1500.0}
```

Note that keys are not normalized here.

Shift keys from UCTE glsk files

Alternatively zones can also be created with weighted injections defined in ucte GLSK files. Two ways of creating zones are available. The first one use a glsk file and create a list of Zone objects with all the areas defined within:

```
>>> n = pp.network.load('simple-eu.uct')
>>> zones = pp.sensitivity.create_zones_from_glsk_file(n, 'glsk_sample.xml', datetime.
↪datetime(2019, 1, 8, 0, 0))
>>> params = pp.loadflow.Parameters(distributed_slack=False)
>>> sa = pp.sensitivity.create_dc_analysis()
>>> sa.set_zones(zones)
>>> sa.add_branch_flow_factor_matrix(['BBE2AA1  FFR3AA1  1'], ['10YCB-GERMANY--8'], 'm')
>>> results = sa.run(n, params)
```

The second one allows a more refined zone creation by separating the glsk file data loading and the zone creation:

```
>>> n = pp.network.load('simple-eu.uct')
>>> glsk_document = pp.glsk.load('glsk_sample.xml')
>>> t_start = glsk_document.get_gsk_time_interval_start()
>>> t_end = glsk_document.get_gsk_time_interval_end()
>>> de_generators = glsk_document.get_points_for_country(n, '10YCB-GERMANY--8', t_start)
>>> de_shift_keys = glsk_document.get_glsk_factors(n, '10YCB-GERMANY--8', t_start)
>>> zone_de = pp.sensitivity.create_zone_from_injections_and_shift_keys('10YCB-GERMANY--8
↪', de_generators, de_shift_keys)
```

Zone modification

You can change a zone perimeter. In the following example, we imagine that the bus 'DDE3AA1' moves from DE zone to FR zone.

```
>>> zone_fr = pp.sensitivity.create_country_zone(n, 'FR')
>>> zone_fr.injections_ids
['FFR1AA1 _generator',
 'FFR2AA1 _generator',
 'FFR3AA1 _generator']
>>> zone_de = pp.sensitivity.create_country_zone(n, 'DE')
>>> zone_de.injections_ids
['DDE1AA1 _generator',
 'DDE2AA1 _generator',
 'DDE3AA1 _generator']
>>> zone_de.move_injection_to(zone_fr, 'DDE3AA1 _generator')
>>> zone_fr.injections_ids
['FFR1AA1 _generator',
 'FFR2AA1 _generator',
```

(continues on next page)

(continued from previous page)

```
 'FFR3AA1 _generator',
 'DDE3AA1 _generator']
>>> zone_de.injections_ids
['DDE1AA1 _generator',
 'DDE2AA1 _generator']
```

If we rerun the sensitivity calculation, we found that 1 MW active power transfer from FR zone to DE zone will be responsible of a variation of -0.239337 MW (previously -0.256641 MW) on the border line 'BBE2AA1 FFRAA1 1'. Changing the monitored branch could be relevant in that use case to simulate that borders have moved.

We can also create an zone totally empty and transfer injections from other country zones to this new one.

```
>>> zone_fict = pp.sensitivity.create_empty_zone('FICT')
>>> zone_fr.move_injection_to(zone_fict, 'DDE3AA1 _generator')
>>> zone_fict.injections_ids
['DDE3AA1 _generator']
```

Other kind of sensitivities

PyPowSyBl allows to compute more that PTDF. In addition to injections and zones you configure the sensitivity matrix with:

- a phase shifter ID to compute the sensitivity of a phase shifting on a branch, feature also called Phase Shift Distribution Factor (PSDF)
- a HVDC line ID if you want to see the effect of an increase of the active power set point on a other branch (better known as DCDF). Note that in that case, the HVDC line must be explicitly described in the network through *HvdcLine* object. If the HVDC line is modeled with two injections because the HVDC line is not explicitly modeled (as in network coming from UCTE format), you have to put both injection ids and make the difference between the sensitivity results.

```
>>> sa.add_branch_flow_factor_matrix(['BBE2AA1  FFR3AA1  1'], [zone, injection_id,␣
↪transformer_id, hvdc_id], 'm')
```

AC sensitivity analysis

It's possible to perform an AC sensitivity analysis almost in the same way, just use `create_ac_analysis` instead of `create_dc_analysis`:

```
>>> analysis = pp.sensitivity.create_ac_analysis()
```

Additionally, AC sensitivity analysis allows to compute voltage sensitivities. You just need to define the list of buses for which you want to compute the sensitivity, and a list of regulating equipments (generators, transformers, etc.):

```
>>> analysis = pp.sensitivity.create_ac_analysis()
>>> analysis.add_bus_voltage_factor_matrix(bus_ids=['VLHV1_0', 'VLLOAD_0'], target_
↪voltage_ids=['GEN'])
>>> result = analysis.run(network)
>>> result.get_sensitivity_matrix()
       VLHV1_0  VLLOAD_0
GEN  17.629602   7.89637
```

Post-contingency analysis

In previous paragraphs, sensitivities were only computed on pre-contingency situation. Additionally, you can compute sensitivities on post-contingency situations, by adding contingency definitions to your analysis:

```
>>> analysis = pp.sensitivity.create_dc_analysis()
>>> analysis.add_branch_flow_factor_matrix(branches_ids=['NHV1_NHV2_1', 'NHV1_NHV2_2'],␣
↪variables_ids=['LOAD'], matrix_id='m')
>>> analysis.add_single_element_contingency('NHV1_NHV2_1')
>>> result = analysis.run(network)
>>> result.get_reference_matrix('m', 'NHV1_NHV2_1')
                  NHV1_NHV2_1  NHV1_NHV2_2
reference_values          NaN        600.0
>>> result.get_sensitivity_matrix('m', 'NHV1_NHV2_1')
      NHV1_NHV2_1  NHV1_NHV2_2
LOAD          0.0         -1.0
```

Pre-contingency only or specific post-contingencies state analysis

You can also limit the computation of your sensitivities to the pre contingency state or to some specific post contingencies states by using add/get precontingency_branch_flow_factor_matrix and postcontingency_branch_flow_factor_matrix methods.

```
>>> analysis = pp.sensitivity.create_dc_analysis()
>>> analysis.add_precontingency_branch_flow_factor_matrix(branches_ids=['NHV1_NHV2_1',
↪'NHV1_NHV2_2'], variables_ids=['LOAD'], matrix_id='precontingency')
>>> analysis.add_postcontingency_branch_flow_factor_matrix(branches_ids=['NHV1_NHV2_1',
↪'NHV1_NHV2_2'], variables_ids=['GEN'], contingencies_ids=['NHV1_NHV2_1'], matrix_id=
↪'postcontingency')
>>> analysis.add_single_element_contingency('NHV1_NHV2_1')
>>> result = analysis.run(network)
>>> result.get_sensitivity_matrix('precontingency')
      NHV1_NHV2_1  NHV1_NHV2_2
LOAD         -0.5         -0.5
>>> result.get_sensitivity_matrix('postcontingency', 'NHV1_NHV2_1')
     NHV1_NHV2_1  NHV1_NHV2_2
GEN          0.0          0.0
```

Advanced sensitivity analysis factors configuration

For advanced users, a more generic way to create factors is available allowing to define the function and the variable type (sensitivity is defined as the derivative of the function with respect to the variable).

```
>>> analysis = pp.sensitivity.create_ac_analysis()
>>> analysis.add_factor_matrix(functions_ids=['NHV1_NHV2_1'], variables_ids=['LOAD'],␣
↪contingency_context_type=pp.sensitivity.ContingencyContextType.NONE, contingencies_
↪ids=[], sensitivity_function_type=pp.sensitivity.SensitivityFunctionType.BRANCH_ACTIVE_
↪POWER_2, sensitivity_variable_type=pp.sensitivity.SensitivityVariableType.INJECTION_
↪ACTIVE_POWER)
>>> result = analysis.run(network)
>>> result.get_sensitivity_matrix()
      NHV1_NHV2_1
LOAD     0.501398
```

As pypowsybl uses powsybl-open-loadflow to compute sensitivity analysis, the list of supported functions and variables in AC or DC is the same as the one documented in the OpenLoadFlow documentation.

A special value of *SensitivityVariableType AUTO_DETECT* allows to auto detect each of the variable type using its ID. It is important to notice that in this case, not all type of sensitivity variable are usable. For instance when an ID of a busbar section is given as a variable and function is a current flow, the detected variable will be an active power injection. This is an arbitrary choice because it could also have been a voltage.

2.1.9 Logging configuration

All PyPowSyBl logs are sent to 'powsybl' logger. This logger is by default configured with a null handler so that none of the logs are printed.

A simple way to see PyPowSyBl logs is to create a basic config and set log level on 'powsybl' logger:

```
import logging
logging.basicConfig()
logging.getLogger('powsybl').setLevel(logging.INFO)
```

A non standard log level with value 1 can be used to get TRACE logs of Java side (logback):

```
logging.getLogger('powsybl').setLevel(1)
```

To specify a more readable log format:

```
logging.basicConfig(format='%(asctime)s - %(levelname)s - %(message)s')
```

2.1.10 Running a flow decomposition

You can use the module *pypowsybl.flowdecomposition* in order to run flow decomposition on networks. Please check out the examples below.

The general idea of this API is to create a decomposition object. Then, you can define contingencies if necessary. Then, you can define XNE and XNEC. XNEC definition requires pre-defined contingencies. Some pre-defined XNE selection adder functions are available. All the adder functions will be united when running a flow decomposition. Finally, you can run the flow decomposition with some flow decomposition and/or load flow parameters.

For detailed documentation of involved classes and methods, please refer to the *API reference*.

Start by importing the module:

```
import pypowsybl as pp
```

First example

To perform a flow decomposition, you need at least a network. We will define a flow decomposition object, add some contingencies and some monitored lines. Those lines will be mapped to the network when running a flow decomposition. The flow decomposition computation returns a data frame containing the flow decomposition and the reference values. The reference values are the active power flows in AC on the original network and in DC on the compensated network. By default, the compensated network is the same as the original network as the loss compensation is not activated by default. Here are toy examples that do not reflect reality.

```
>>> network = pp.network.create_eurostag_tutorial_example1_network()
>>> branch_ids = ['NHV1_NHV2_1', 'NHV1_NHV2_2']
>>> flow_decomposition = pp.flowdecomposition.create_decomposition() \
...     .add_single_element_contingencies(branch_ids) \
```

(continues on next page)

(continued from previous page)

```
...     .add_monitored_elements(branch_ids, branch_ids)
>>> flow_decomposition_dataframe = flow_decomposition.run(network)
>>> flow_decomposition_dataframe
                          branch_id contingency_id country1 country2  ac_reference_
↪flow1  ac_reference_flow2  dc_reference_flow  commercial_flow  x_node_flow  pst_flow ␣
↪internal_flow  loop_flow_from_be  loop_flow_from_fr
xnec_id
NHV1_NHV2_1              NHV1_NHV2_1                      FR       BE          302.
↪444049         -300.433895              300.0             0.0          0.0       0.0 ␣
↪          0.0              300.0               0.0
NHV1_NHV2_1_NHV1_NHV2_2  NHV1_NHV2_1    NHV1_NHV2_2       FR       BE          610.
↪562161         -600.996158              600.0             0.0          0.0       0.0 ␣
↪          0.0              600.0               0.0
NHV1_NHV2_2              NHV1_NHV2_2                      FR       BE          302.
↪444049         -300.433895              300.0             0.0          0.0       0.0 ␣
↪          0.0              300.0               0.0
NHV1_NHV2_2_NHV1_NHV2_1  NHV1_NHV2_2    NHV1_NHV2_1       FR       BE          610.
↪562161         -600.996158              600.0             0.0          0.0       0.0 ␣
↪          0.0              600.0               0.0
```

Loop flows

Here is another example with imbricated zones. This example will highlight loop flows from the peripheral areas.

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_LOOP_FLOW_WITH_COUNTRIES.uct
↪')))
>>> flow_decomposition = pp.flowdecomposition.create_decomposition().add_monitored_
↪elements(['BLOAD 11 FLOAD 11 1', 'EGEN  11 FGEN  11 1', 'FGEN  11 BGEN  11 1', 'FLOAD␣
↪11 ELOAD 11 1'])
>>> flow_decomposition_dataframe = flow_decomposition.run(network)
>>> flow_decomposition_dataframe
                            branch_id contingency_id country1 country2  ac_reference_
↪flow1  ac_reference_flow2  dc_reference_flow  commercial_flow  x_node_flow  pst_flow ␣
↪internal_flow  loop_flow_from_be  loop_flow_from_es  loop_flow_from_fr
xnec_id
BLOAD 11 FLOAD 11 1  BLOAD 11 FLOAD 11 1                      BE       FR             ␣
↪ NaN                NaN             200.0     0.000000e+00          0.0       0.0   ␣
↪        0.0       0.000000e+00             100.0       1.000000e+02
EGEN  11 FGEN  11 1  EGEN  11 FGEN  11 1                      ES       FR             ␣
↪ NaN                NaN             100.0    -8.526513e-14          0.0       0.0   ␣
↪        0.0       3.552714e-14             100.0      -3.552714e-14
FGEN  11 BGEN  11 1  FGEN  11 BGEN  11 1                      FR       BE             ␣
↪ NaN                NaN             200.0    -1.421085e-13          0.0       0.0   ␣
↪        0.0       8.526513e-14             100.0       1.000000e+02
FLOAD 11 ELOAD 11 1  FLOAD 11 ELOAD 11 1                      FR       ES             ␣
↪ NaN                NaN             100.0     0.000000e+00          0.0       0.0   ␣
↪        0.0       0.000000e+00             100.0       0.000000e+00
```

On this example, the AC load flow does not converge, the fallback to DC load flow is activated by default. This means that the AC reference values are NaNs. For each line where the AC reference is not a number, the rescaling is disabled to prevent NaN propagation.

PST flows

Network details

Here is another example with a more complex network containing a phase-shifting transformer (PST). This PST has a non-neutral tap position, thus forcing the flows in a certain direction. This example illustrates the flow decomposition with such network element.

As we cannot set a PST on an interconnection, we set an equivalent null load called 'BLOAD 11'.

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_PST_FLOW_WITH_COUNTRIES.uct
↪')))
>>> network.get_generators()
                  name energy_source  target_p   min_p   max_p   min_q   max_q  rated_s␣
↪reactive_limits_kind  target_v  target_q  voltage_regulator_on regulated_element_id  ␣
↪p   q   i voltage_level_id     bus_id  connected
id
FGEN  11_generator              OTHER     100.0 -1000.0  1000.0 -1000.0  1000.0      NaN␣
↪             MIN_MAX     400.0       0.0                  True   FGEN  11_generator␣
↪NaN NaN NaN          FGEN  1  FGEN  1_0       True
BLOAD 12_generator              OTHER     100.0 -1000.0  1000.0 -1000.0  1000.0      NaN␣
↪             MIN_MAX     400.0       0.0                  True   BLOAD 12_generator␣
↪NaN NaN NaN          BLOAD 1  BLOAD 1_1       True
>>> network.get_loads()
                 name       type     p0   q0   p   q   i voltage_level_id     bus_id ␣
↪connected
id
BLOAD 12_load       UNDEFINED  200.0  0.0 NaN NaN NaN          BLOAD 1  BLOAD 1_1      ␣
↪True
>>> network.get_lines()
                      name    r    x   g1   b1   g2   b2  p1  q1  i1  p2  q2  i2␣
↪voltage_level1_id voltage_level2_id    bus1_id    bus2_id  connected1  connected2
id
FGEN  11 BLOAD 12 1       0.5  1.5  0.0  0.0  0.0  0.0 NaN NaN NaN NaN NaN NaN          ␣
↪FGEN  1           BLOAD 1  FGEN  1_0  BLOAD 1_1        True        True
FGEN  11 BLOAD 11 1       1.0  3.0  0.0  0.0  0.0  0.0 NaN NaN NaN NaN NaN NaN          ␣
↪FGEN  1           BLOAD 1  FGEN  1_0  BLOAD 1_0        True        True
>>> network.get_buses()
             name  v_mag  v_angle  connected_component  synchronous_component voltage_
↪level_id
id
FGEN  1_0         NaN      NaN                    0                     0          FGEN␣
↪ 1
BLOAD 1_0         NaN      NaN                    0                     0        ␣
↪BLOAD 1
BLOAD 1_1         NaN      NaN                    0                     0        ␣
↪BLOAD 1
>>> network.get_2_windings_transformers()
                      name    r    x       g        b  rated_u1  rated_u2  rated_s  p1␣
↪ q1  i1  p2  q2  i2 voltage_level1_id voltage_level2_id    bus1_id    bus2_id ␣
↪connected1  connected2
id
BLOAD 11 BLOAD 12 2       0.5  1.5  0.0002  0.00015     400.0     400.0      NaN NaN NaN␣
↪NaN NaN NaN NaN           BLOAD 1           BLOAD 1  BLOAD 1_1  BLOAD 1_0        True ␣
```

(continues on next page)

(continued from previous page)

```
↪      True
>>> network.get_phase_tap_changers()
                   side  tap  solved_tap_position low_tap  high_tap  step_count oltc␣
↪regulating  regulation_mode  regulation_value  target_deadband regulating_bus_id
id
BLOAD 11 BLOAD 12 2         0                  NaN     -16        16          33 True   ␣
↪   False  CURRENT_LIMITER              NaN              NaN
```

Neutral tap position

Here are the results with neutral tap position.

```
>>> flow_decomposition = pp.flowdecomposition.create_decomposition().add_monitored_
↪elements(['FGEN  11 BLOAD 11 1', 'FGEN  11 BLOAD 12 1'])
>>> flow_decomposition_dataframe = flow_decomposition.run(network)
>>> flow_decomposition_dataframe
                             branch_id contingency_id country1 country2  ac_reference_
↪flow1  ac_reference_flow2  dc_reference_flow  commercial_flow  x_node_flow  pst_flow ␣
↪internal_flow  loop_flow_from_be  loop_flow_from_fr
xnec_id
FGEN  11 BLOAD 11 1  FGEN  11 BLOAD 11 1                     FR       BE           29.
↪003009          -28.997170               25.0        28.999015          0.0      -0.0 ␣
↪          0.0          -1.999508          -1.999508
FGEN  11 BLOAD 12 1  FGEN  11 BLOAD 12 1                     FR       BE           87.
↪009112          -86.982833               75.0        86.997046          0.0       0.0 ␣
↪          0.0          -5.998523          -5.998523
>>> flow_decomposition_dataframe[[c for c in flow_decomposition_dataframe.columns if (
↪"flow" in c and "reference" not in c)]].sum(axis=1)
xnec_id
FGEN  11 BLOAD 11 1    25.0
FGEN  11 BLOAD 12 1    75.0
dtype: float64
```

The results are not rescaled to the AC reference by default.

Non neutral tap position

Here are the results with non-neutral tap position.

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_PST_FLOW_WITH_COUNTRIES.uct
↪')))
>>> network.update_phase_tap_changers(id="BLOAD 11 BLOAD 12 2", tap=1)
>>> network.get_phase_tap_changers()
                   side  tap  solved_tap_position low_tap  high_tap  step_count  oltc␣
↪regulating regulation_mode  regulation_value  target_deadband regulating_bus_id
id
BLOAD 11 BLOAD 12 2         1                  NaN     -16        16          33  True  ␣
↪   False CURRENT_LIMITER              NaN              NaN
>>> flow_decomposition = pp.flowdecomposition.create_decomposition().add_monitored_
↪elements(['FGEN  11 BLOAD 11 1', 'FGEN  11 BLOAD 12 1'])
>>> flow_decomposition_dataframe = flow_decomposition.run(network)
>>> flow_decomposition_dataframe
```

(continues on next page)

(continued from previous page)

```
                              branch_id contingency_id country1 country2  ac_reference_
↪flow1  ac_reference_flow2  dc_reference_flow  commercial_flow  x_node_flow    pst_flow␣
↪ internal_flow  loop_flow_from_be  loop_flow_from_fr
xnec_id
FGEN  11 BLOAD 11 1  FGEN  11 BLOAD 11 1                      FR       BE          192.
↪390656         -192.134125         188.652703        29.015809          0.0  163.
↪652703            0.0          -2.007905          -2.007905
FGEN  11 BLOAD 12 1  FGEN  11 BLOAD 12 1                      FR       BE          -76.
↪189072           76.209233         -88.652703       -87.047428          0.0  163.
↪652703            0.0           6.023714           6.023714
>>> flow_decomposition_dataframe[[c for c in flow_decomposition_dataframe.columns if (
↪"flow" in c and "reference" not in c)]].sum(axis=1)
xnec_id
FGEN  11 BLOAD 11 1    188.652703
FGEN  11 BLOAD 12 1     88.652703
dtype: float64
```

Note that the reference flow on the 2d branch has changed of sign. As we use it as reference, all the decomposed flows have also changed of sign.

Unmerged X node flows

To illustrate X node flow, we need a network with unmerged x nodes. Those x nodes might represent HVDCs, outside countries, etc. Merged X nodes will not be considered here.

```
>>> network = pp.network.load(DATA_DIR.joinpath('19700101_0000_FO4_UX1.uct'))
>>> flow_decomposition = pp.flowdecomposition.create_decomposition().add_
↪interconnections_as_monitored_elements()
>>> flow_decomposition.run(network)
                                                                                 branch_id␣
↪contingency_id country1 country2  ac_reference_flow1  ac_reference_flow2  dc_reference_
↪flow  commercial_flow  x_node_flow  pst_flow  internal_flow  loop_flow_from_be  loop_
↪flow_from_de  loop_flow_from_fr
xnec_id
XBD00011 BD000011 1 + XBD00011 DB000011 1  XBD00011 BD000011 1 + XBD00011 DB000011 1    ␣
↪                  BE       DE             121.822            -121.822            124.
↪685          171.517      -33.155     2.952          0.000              0.226         ␣
↪   -0.000            -16.854
XBD00012 BD000011 1 + XBD00012 DB000011 1  XBD00012 BD000011 1 + XBD00012 DB000011 1    ␣
↪                  BE       DE             121.822            -121.822            124.
↪685          171.517      -33.155     2.952          0.000              0.226         ␣
↪   -0.000            -16.854
XBF00011 BF000011 1 + XBF00011 FB000011 1  XBF00011 BF000011 1 + XBF00011 FB000011 1    ␣
↪                  BE       FR            -775.578             775.578           -764.
↪445          679.262      170.472     7.112          0.000           -124.053         ␣
↪   -0.000             31.652
XBF00021 BF000021 1 + XBF00021 FB000021 1  XBF00021 BF000021 1 + XBF00021 FB000021 1    ␣
↪                  BE       FR            -234.033             234.033           -242.
↪463          169.386       44.108    -0.604          0.000             62.253         ␣
↪   -0.000            -32.680
XBF00022 BF000021 1 + XBF00022 FB000022 1  XBF00022 BF000021 1 + XBF00022 FB000022 1    ␣
↪                  BE       FR            -234.033             234.033           -242.
```

(continues on next page)

(continued from previous page)

```
↪463          169.386       44.108    -0.604          0.000             62.253          ␣
↪   -0.000            -32.680
XDF00011 DF000011 1 + XDF00011 FD000011 1  XDF00011 DF000011 1 + XDF00011 FD000011 1    ␣
↪                 DE       FR          -1,156.356          1,156.356         -1,150.
↪629          906.966      216.311    -5.903          0.000             -0.453          ␣
↪   -0.000             33.709
```

Adder functions

The flow decomposition algorithm will decompose flow on monitored elements. You need to define those elements. You can either define those elements with specific ids or with automatic functions.

The union of selected elements will be decomposed. For example, if you select the same branch in the same state two times, it will be decomposed only once.

Specific adder functions

Specific adder functions are based on IDs. When running the flow decomposition, the IDs will be mapped to the network. If an identifiable is not found on the network, a warning will be sent (beware of activated logs) and the corresponding XNEC will be ignored.

With those adder functions, you can create XNEs and/or XNECs. You need to specify contingencies first if required. If you try to create a XNEC with an undefined contingency ID, an error will be raised.

By default, if you add monitored elements with branches and contingencies, it will create all possible valid pairs of branch and states. By default, all the states are base case and all contingency states defined. You can specify which states you want in the base add monitored element function or use a dedicated pre/post contingency function.

Here is an example

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_PST_FLOW_WITH_COUNTRIES.uct
↪')))
>>> flow_decomposition = pp.flowdecomposition.create_decomposition() \
... .add_monitored_elements(['FGEN  11 BLOAD 11 1']) \
... .add_single_element_contingency('FGEN  11 BLOAD 11 1') \
... .add_monitored_elements(['FGEN  11 BLOAD 12 1'], ['FGEN  11 BLOAD 11 1']) \
... .add_multiple_elements_contingency(['FGEN  11 BLOAD 11 1', 'BLOAD 11 BLOAD 12 2']) \
... .add_monitored_elements('FGEN  11 BLOAD 12 1', 'FGEN  11 BLOAD 11 1_BLOAD 11 BLOAD␣
↪12 2', pp.flowdecomposition.ContingencyContextType.SPECIFIC)
>>> flow_decomposition.run(network)
                                                          branch_id                 ␣
↪         contingency_id country1 country2  ac_reference_flow1  ac_reference_flow2  dc_
↪reference_flow  commercial_flow  x_node_flow  pst_flow  internal_flow  loop_flow_from_
↪be  loop_flow_from_fr
xnec_id
FGEN  11 BLOAD 11 1                                FGEN  11 BLOAD 11 1               ␣
↪                              FR       BE           29.003009         -28.997170     ␣
↪         25.0        28.999015          0.0      -0.0            0.0          -1.
↪999508          -1.999508
FGEN  11 BLOAD 12 1                                FGEN  11 BLOAD 12 1               ␣
↪                              FR       BE           87.009112         -86.982833     ␣
↪         75.0        86.997046          0.0       0.0            0.0          -5.
↪998523          -5.998523
FGEN  11 BLOAD 12 1_FGEN  11 BLOAD 11 1            FGEN  11 BLOAD 12 1               ␣
```

(continues on next page)

(continued from previous page)

```
↪    FGEN  11 BLOAD 11 1       FR       BE          116.016179         -115.969462     ␣
↪         100.0       115.996062          0.0       0.0            0.0          -7.
↪998031          -7.998031
FGEN  11 BLOAD 12 1_FGEN  11 BLOAD 11 1_BLOAD 1...  FGEN  11 BLOAD 12 1  FGEN  11 BLOAD␣
↪11 1_BLOAD 11 BLOAD 12 2       FR       BE          100.034531          -99.999797    ␣
↪         100.0       115.996062          0.0       0.0            0.0          -7.
↪998031          -7.998031
```

See the API reference for more details about how each specific adder works.

Automatic adder functions

Automatic adder functions are based on automatic selection processes. With those functions, you can create XNEs and/or XNECs.

Some automatic XNE selection adder functions are available.

5% zonal PTDF criteria

This adder function will add all branches in the N state that have a zone-to-zone PTDF greater than 5% or that are interconnections. This function adds some non-negligible precomputing to the process.

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_PST_FLOW_WITH_COUNTRIES.uct
↪')))
>>> flow_decomposition = pp.flowdecomposition.create_decomposition() \
... .add_5perc_ptdf_as_monitored_elements()
>>> flow_decomposition.run(network)
                             branch_id contingency_id country1 country2  ac_reference_
↪flow1  ac_reference_flow2  dc_reference_flow  commercial_flow  x_node_flow  pst_flow ␣
↪internal_flow  loop_flow_from_be  loop_flow_from_fr
xnec_id
BLOAD 11 BLOAD 12 2  BLOAD 11 BLOAD 12 2                     BE       BE           3.
↪005666           28.997253             -25.0        28.999015          0.0      -0.0 ␣
↪    -1.999508           0.000000          -1.999508
FGEN  11 BLOAD 11 1  FGEN  11 BLOAD 11 1                     FR       BE          29.
↪003009          -28.997170              25.0        28.999015          0.0      -0.0 ␣
↪     0.000000          -1.999508          -1.999508
FGEN  11 BLOAD 12 1  FGEN  11 BLOAD 12 1                     FR       BE          87.
↪009112          -86.982833              75.0        86.997046          0.0       0.0 ␣
↪     0.000000          -5.998523          -5.998523
```

Interconnections

This adder function will add interconnections in the N state. Be careful when using this function with large networks.

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_PST_FLOW_WITH_COUNTRIES.uct
↪')))
>>> flow_decomposition = pp.flowdecomposition.create_decomposition() \
... .add_interconnections_as_monitored_elements()
>>> flow_decomposition.run(network)
                             branch_id contingency_id country1 country2  ac_reference_
↪flow1  ac_reference_flow2  dc_reference_flow  commercial_flow  x_node_flow  pst_flow ␣
```

(continues on next page)

(continued from previous page)

```
↪internal_flow  loop_flow_from_be  loop_flow_from_fr
xnec_id
FGEN  11 BLOAD 11 1  FGEN  11 BLOAD 11 1                    FR       BE           29.
↪003009          -28.997170               25.0        28.999015          0.0      -0.0 ␣
↪          0.0          -1.999508          -1.999508
FGEN  11 BLOAD 12 1  FGEN  11 BLOAD 12 1                    FR       BE           87.
↪009112          -86.982833               75.0        86.997046          0.0       0.0 ␣
↪          0.0          -5.998523          -5.998523
```

All branches

This adder function will add all branches in the N state. Be careful when using this function with large networks.

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_PST_FLOW_WITH_COUNTRIES.uct
↪')))
>>> flow_decomposition = pp.flowdecomposition.create_decomposition() \
... .add_all_branches_as_monitored_elements()
>>> flow_decomposition.run(network)
                            branch_id contingency_id country1 country2  ac_reference_
↪flow1  ac_reference_flow2  dc_reference_flow  commercial_flow  x_node_flow  pst_flow ␣
↪internal_flow  loop_flow_from_be  loop_flow_from_fr
xnec_id
BLOAD 11 BLOAD 12 2  BLOAD 11 BLOAD 12 2                    BE       BE            3.
↪005666           28.997253              -25.0        28.999015          0.0      -0.0 ␣
↪    -1.999508           0.000000          -1.999508
FGEN  11 BLOAD 11 1  FGEN  11 BLOAD 11 1                    FR       BE           29.
↪003009          -28.997170               25.0        28.999015          0.0      -0.0 ␣
↪     0.000000          -1.999508          -1.999508
FGEN  11 BLOAD 12 1  FGEN  11 BLOAD 12 1                    FR       BE           87.
↪009112          -86.982833               75.0        86.997046          0.0       0.0 ␣
↪     0.000000          -5.998523          -5.998523
```

Mixing adder functions

You can mix everything together as you like.

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_PST_FLOW_WITH_COUNTRIES.uct
↪')))
>>> parameters = pp.flowdecomposition.Parameters(sensitivity_epsilon=pp.
↪flowdecomposition.Parameters.DISABLE_SENSITIVITY_EPSILON)
>>> flow_decomposition = pp.flowdecomposition.create_decomposition() \
... .add_single_element_contingency('FGEN  11 BLOAD 11 1') \
... .add_monitored_elements(['FGEN  11 BLOAD 12 1', 'BLOAD 11 BLOAD 12 2'], ['FGEN  11␣
↪BLOAD 11 1']) \
... .add_multiple_elements_contingency(['FGEN  11 BLOAD 11 1', 'BLOAD 11 BLOAD 12 2']) \
... .add_postcontingency_monitored_elements('FGEN  11 BLOAD 12 1', 'FGEN  11 BLOAD 11 1_
↪BLOAD 11 BLOAD 12 2') \
... .add_interconnections_as_monitored_elements() \
... .add_all_branches_as_monitored_elements()
>>> flow_decomposition.run(network, flow_decomposition_parameters=parameters)
                                                            branch_id                ␣
```

(continues on next page)

(continued from previous page)

```
↪         contingency_id country1 country2  ac_reference_flow1  ac_reference_flow2  dc_
↪reference_flow  commercial_flow  x_node_flow  pst_flow  internal_flow  loop_flow_from_
↪be  loop_flow_from_fr
xnec_id
BLOAD 11 BLOAD 12 2                                  BLOAD 11 BLOAD 12 2                ␣
↪                                BE       BE            3.005666           28.997253     ␣
↪        -25.0        28.999015          0.0      -0.0      -1.999508           0.
↪000000          -1.999508
BLOAD 11 BLOAD 12 2_FGEN  11 BLOAD 11 1              BLOAD 11 BLOAD 12 2                ␣
↪    FGEN  11 BLOAD 11 1       BE       BE           32.000000            0.000000     ␣
↪         -0.0         0.000000          0.0       0.0       0.000000           0.
↪000000          -0.000000
FGEN  11 BLOAD 11 1                                  FGEN  11 BLOAD 11 1                ␣
↪                                FR       BE           29.003009          -28.997170     ␣
↪         25.0        28.999015          0.0      -0.0       0.000000          -1.
↪999508          -1.999508
FGEN  11 BLOAD 12 1                                  FGEN  11 BLOAD 12 1                ␣
↪                                FR       BE           87.009112          -86.982833     ␣
↪         75.0        86.997046          0.0       0.0       0.000000          -5.
↪998523          -5.998523
FGEN  11 BLOAD 12 1_FGEN  11 BLOAD 11 1              FGEN  11 BLOAD 12 1                ␣
↪    FGEN  11 BLOAD 11 1       FR       BE          116.016179         -115.969462     ␣
↪        100.0       115.996062          0.0       0.0       0.000000          -7.
↪998031          -7.998031
FGEN  11 BLOAD 12 1_FGEN  11 BLOAD 11 1_BLOAD 1...  FGEN  11 BLOAD 12 1  FGEN  11 BLOAD␣
↪11 1_BLOAD 11 BLOAD 12 2       FR       BE           100.034531          -99.999797     ␣
↪         100.0       115.996062          0.0       0.0       0.000000          -7.
↪998031          -7.998031
```

Note: if one of our xnec is missing, it might be caused by a zero MW DC reference flow, you can show them by reducing the sensitivity-epsilon as bone before. This will be fixed in next versions.

Configuration file

Inside your config.yml file, you can change the default Configuration of the flow decomposition. Here are the available parameters and their default values:

Listing 1: Available parameters and their default values

```
flow-decomposition-default-parameters:
    enable-losses-compensation: False
    losses-compensation-epsilon: 1e-5
    sensitivity-epsilon: 1e-5
    rescale-mode: ACER_METHODOLOGY
    dc-fallback-enabled-after-ac-divergence: True
    sensitivity-variable-batch-size: 15000
```

The flow decomposition parameters can be overwritten in Python. If you have memory issues, do not hesitate to reduce the *sensitivity-variable-batch-size* parameter.

```
>>> network = pp.network.load(str(DATA_DIR.joinpath('NETWORK_PST_FLOW_WITH_COUNTRIES.uct
↪')))
>>> parameters = pp.flowdecomposition.Parameters(enable_losses_compensation=True,
```

(continues on next page)

(continued from previous page)

```
... losses_compensation_epsilon=pp.flowdecomposition.Parameters.DISABLE_LOSSES_
↪COMPENSATION_EPSILON,
... sensitivity_epsilon=pp.flowdecomposition.Parameters.DISABLE_SENSITIVITY_EPSILON,
... rescale_mode=pp.flowdecomposition.RescaleMode.ACER_METHODOLOGY,
... dc_fallback_enabled_after_ac_divergence=True,
... sensitivity_variable_batch_size=1000)
>>> flow_decomposition = pp.flowdecomposition.create_decomposition().add_monitored_
↪elements(['BLOAD 11 BLOAD 12 2', 'FGEN  11 BLOAD 11 1', 'FGEN  11 BLOAD 12 1'])
>>> flow_decomposition_dataframe = flow_decomposition.run(network, parameters)
>>> flow_decomposition_dataframe
                            branch_id contingency_id country1 country2  ac_reference_
↪flow1  ac_reference_flow2  dc_reference_flow  commercial_flow  x_node_flow  pst_flow ␣
↪internal_flow  loop_flow_from_be  loop_flow_from_fr
xnec_id
BLOAD 11 BLOAD 12 2  BLOAD 11 BLOAD 12 2                     BE       BE            3.
↪005666           28.997253          -4.994160        27.010522          0.0      -0.0 ␣
↪   -24.003523           0.000000          -0.001333
FGEN  11 BLOAD 11 1  FGEN  11 BLOAD 11 1                     FR       BE           29.
↪003009          -28.997170          36.997080        22.733336          0.0      -0.0 ␣
↪     0.000000           6.271006          -0.001333
FGEN  11 BLOAD 12 1  FGEN  11 BLOAD 12 1                     FR       BE           87.
↪009112          -86.982833          78.988321        95.017838          0.0       0.0 ␣
↪     0.000000          -8.004728          -0.003998
```

You can also overwrite the Load flow parameters.

```
>>> network = pp.network.create_eurostag_tutorial_example1_network()
>>> flow_decomposition_parameters = pp.flowdecomposition.Parameters()
>>> load_flow_parameters = pp.loadflow.Parameters()
>>> flow_decomposition = pp.flowdecomposition.create_decomposition().add_monitored_
↪elements(['NHV1_NHV2_1', 'NHV1_NHV2_2'])
>>> flow_decomposition_dataframe = flow_decomposition.run(network, flow_decomposition_
↪parameters, load_flow_parameters)
>>> flow_decomposition_dataframe
              branch_id contingency_id country1 country2  ac_reference_flow1  ac_
↪reference_flow2  dc_reference_flow  commercial_flow  x_node_flow  pst_flow  internal_
↪flow  loop_flow_from_be  loop_flow_from_fr
xnec_id
NHV1_NHV2_1  NHV1_NHV2_1                       FR       BE          302.444049         -
↪300.433895              300.0              0.0          0.0       0.0            0.0  ␣
↪           300.0                 0.0
NHV1_NHV2_2  NHV1_NHV2_2                       FR       BE          302.444049         -
↪300.433895              300.0              0.0          0.0       0.0            0.0  ␣
↪           300.0                 0.0
```

2.1.11 Running a dynamic simulation with dynawo

You can use the module *pypowsybl.dynamic* in order to run time domain simulation on networks.

Start by importing the module:

```
import pypowsybl.network as pn
import pypowsybl.dynamic as dyn
```

Providers

For now we only support the Dynawo simulator integration, provided by the Dynawo project.

Prerequisites

The pypowsybl config file (generally located at ~/.itools/config.yml) must define the dynawo section to find your dynawo installation and defaults parameters Here is an example of a simple config.yml file. It uses the same configurations as in powsybl-dynawo. Note that parameters can be set programmatically with the *Parameters* class.

```
dynamic-simulation-default-parameters:
    startTime: 0
    stopTime: 30
dynawo:
    homeDir: PATH_TO_DYNAWO
    debug: true
dynawo-simulation-default-parameters:
    parametersFile: ./models.par
    network.parametersFile: ./network.par
    network.parametersId: "1"
    solver.type: IDA
    solver.parametersFile: ./solver.par
    solver.parametersId: "1"
```

Parameters

To make a dynamic simulation, you need multiple things:

1. A dynamic mapping, it links the static elements (generators, loads, lines) to their dynamic behavior (alpha beta load)
2. A event mapping, it maps the different events. (e.g equipment disconnection)
3. A output variable mapping, it records the given values to be watch by the simulation tool (can be curves or final state values).

There is a class for each of these elements.

You will see a lot of arguments called parameterSetId. Dynawo simulator use a lot of parameters that will be stored in files.

Pypowsybl will find the path to this file in the powsybl config.yml in dynawo-simulation-default-parameters.parametersFile value or in *Parameters*.

The parameterSetId argument must match an id in this file (generally called models.par).

Simple example

To run a Dynawo simulation:

```
import pypowsybl.dynamic as dyn
import pypowsybl as pp
from pandas import DataFrame

# load a network
network = pp.network.create_eurostag_tutorial_example1_network()

# dynamic mapping
```

(continues on next page)

(continued from previous page)

```
model_mapping = dyn.ModelMapping()

# get dynamic model categories dataframe
model_mapping.get_categories_information()
# get all supported model information dataframe
model_mapping.get_supported_models_information()
# or filtered by category
model_mapping.get_supported_models_information('SynchronizedGenerator')

# can be written with kwargs...
model_mapping.add_base_load(static_id='LOAD',
                            parameter_set_id='LAB',
                            model_name='LoadAlphaBeta') # and so on
# or dataframe
gen_df = DataFrame.from_records(
    index='static_id',
    columns=['static_id', 'parameter_set_id', 'model_name'],
    data=[('GEN', 'GENPV', 'GeneratorPV')])
model_mapping.add_synchronized_generator(gen_df)
# can also be created with the proper category
model_mapping.add_dynamic_model(category_name='SynchronizedGenerator'
                                static_id='GEN2',
                                parameter_set_id='GENPQ',
                                model_name='GeneratorPQ')

# events mapping
event_mapping = dyn.EventMapping()
event_mapping.add_disconnection(static_id='GEN', start_time=10)
# can also be created with the proper event name
event_mapping.add_event_model(event_name='Disconnect', static_id='NHV1_NHV2_1', start_
↪time=10, disconnect_only='ONE')

# curves mapping
variables_mapping = dyn.OutputVariableMapping()
variables_mapping.add_curves("LOAD", ["load_PPu", "load_QPu"])
variables_mapping.add_final_state_values('NGEN', 'Upu_value') # and so on

# simulations parameters
parameters = dyn.Parameters(start_time=0, stop_time=50)
sim = dyn.Simulation()
# running the simulation
results = sim.run(network, model_mapping, event_mapping, variables_mapping, parameters)
# getting the results
results.status()
results.status_text() # error description if the simulation fails
results.curves() # dataframe containing the mapped curves
results.final_state_values() # dataframe containing the mapped final state values
results.timeline() # dataframe containing the simulation timeline
```

2.1.12 Running a short-circuit analysis

You can use the module *pypowsybl.shortcircuit* in order to perform a shortcircuit analysis on a network. Please have a look at the examples below.

For detailed documentation of involved classes and methods, please refer to the *API reference*.

Note that pypowsybl currently does not include a simulator to perform short-circuit analyses.

Short-circuit analysis

The current APIs allow the simulation of three-phase faults on buses or branches.

To perform a short-circuit analysis, you need a network and at least one fault to simulate on that network. The network should have a transient or subtransient reactance on at least one generator. This reactance is stored in the 'generatorShortCircuit' extension. The results of the analysis include the calculated currents and voltages on the network after the fault. Depending on parameters, the results will be given as three-phase magnitude or detailed on each phase. Optionally, depending on specific parameters of the simulation, the results also include

- the contributions of each feeder to the short-circuit current
- a list of all the violations after the fault
- the voltages, that are higher than a threshold, calculated after the fault on the whole network

Available parameters

The parameters available to run a shortcircuit analysis are:

- with_fortescue_result: indicates whether the currents and voltages are to be given in three-phase magnitude or detailed with magnitude and angle on each phase. This parameter also applies to the feeder results and voltage results.
- with_feeder_result: indicates whether the contributions of each feeder to the short circuit current at the fault node should be calculated.
- with_limit_violations: indicates whether limit violations should be returned after the calculation. If true, a list of buses where the calculated shortcircuit current is higher than the maximum admissible current (stored in ip_max in the identifiableShortCircuit extension) or lower than the minimum admissible current (stored in ip_min in the identifiableShortCircuit extension).
- with_voltage_result: indicates whether the voltage profile should be calculated on every node of the network
- min_voltage_drop_proportional_threshold: specifies a threshold for filtering the voltage results. Only nodes where the voltage drop due to the short circuit is greater than this property are retained.
- study_type: specifies the type of short circuit study. It can be SUB_TRANSIENT, TRANSIENT or STEADY_STATE.
- initial_voltage_profile_mode: specifies the voltage profile to be used for the calculation. It can be either *NOMINAL*, in which case the nominal voltages are used, or *PREVIOUS_VALUE*, in which case the calculated voltages are used.

Parameter	*Default value*
with_fortescue_result	false
with_feeder_result	true
with_limit_violations	true
with_voltage_result	true
min_voltage_drop_proportional_threshold	0
study_type	TRANSIENT

Faults

Faults can be defined either on buses or on branches. The fault resistance and reactance, if specified, are connected in series to ground. In the case of faults on branches, then the location of the fault should be specified, in percent between the two sides of the branch, with the reference to the side 1.

The default values for the fault characteristics are:

Attribute	*Default value*
r	0
x	0
proportional_location (for branch faults)	50

Simple example

```
>>> import pypowsybl as pp
>>> import pypowsybl.network as pn
>>> import pandas as pd
>>> # create a network
>>> n = pn.create_four_substations_node_breaker_network()
>>> # sets some short-circuit parameters
>>> pars = pp.shortcircuit.Parameters(with_feeder_result = False, with_limit_violations␣
↪= False, study_type = pp.shortcircuit.ShortCircuitStudyType.TRANSIENT)
>>> # create a short-circuit analysis context
>>> sc = pp.shortcircuit.create_analysis()
>>> # create a bus fault on the first two buses
>>> buses = n.get_buses()
>>> branches = n.get_branches()
>>> sc.set_faults(id = ['fault_1', 'fault_2'], element_id = [buses.index[0], branches.
↪index[0]], r = [1, 1], x = [2, 2], )
>>> # perform the short-circuit analysis
>>> # results = sc.run(n, pars, 'sc_provider_1')
>>> # returns the analysis results
>>> # results.fault_results
```

2.1.13 Run the voltage initializer

Prerequisites

For now the voltage initializer tool rely on Ampl and Knitro. the binary knitroampl must be in your PATH.

The pypowsybl config file (generally located at ~/.itools/config.yml) must define the ampl section to find your dynawo installation and defaults parameters Here is an example of a simple config.yml file.

```
ampl:
    homeDir: PATH_TO_AMPL
```

Quick start

Here is a simple starting example:

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
```

(continues on next page)

(continued from previous page)

```
params = v_init.VoltageInitializerParameters()
n = pp.network.create_eurostag_tutorial_example1_network()
some_gen_id = n.get_generators().iloc[0].name
params.add_constant_q_generators([some_gen_id])
some_2wt_id = n.get_2_windings_transformers().iloc[0].name
params.add_variable_two_windings_transformers([some_2wt_id])

params.set_objective(VoltageInitializerObjective.SPECIFIC_VOLTAGE_PROFILE)

results = v_init.run(n, params)
results.apply_all_modification(n)

print(results.status())
print(results.indicators())
```

Available settings in the VoltageInitializerParameters class

- Specify which buses will have reactive slacks attached in the ACOPF solving.

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
params.set_reactive_slack_buses_mode(va.VoltageInitializerReactiveSlackBusesMode.NO_
↪GENERATION)
```

- Specify what is the log level of the AMPL solving.

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
params.set_log_level_ampl(va.VoltageInitializerLogLevelAmpl.ERROR)
params.set_log_level_solver(va.VoltageInitializerLogLevelSolver.EVERYTHING)
```

- Change plausible voltage level limits in ACOPF solving.

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
params.set_min_plausible_low_voltage_limit(0.45)
params.set_max_plausible_high_voltage_limit(1.2)
```

- Tune the threshold defining null values in AMPL.

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
params.set_min_plausible_active_power_threshold(1)
params.set_low_impedance_threshold(1e-5)
```

- Modify the parameters used for the correction of generator limits.

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
```

(continues on next page)

(continued from previous page)

```
params = v_init.VoltageInitializerParameters()
params.set_max_plausible_power_limit(7800)
params.set_high_active_power_default_limit(950)
params.set_low_active_power_default_limit(0.5)
params.set_default_minimal_qp_range(0.45)
params.set_default_qmax_pmax_ratio(0.45)
```

- Tune the thresholds used to ignore buses or voltage level limits with nominal voltage lower than them.

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
params.set_min_nominal_voltage_ignored_bus(0.5)
params.set_min_nominal_voltage_ignored_voltage_bounds(1)
```

- Specify which parameters will be variable or fixed in the ACOPF solving

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
n = pp.network.create_eurostag_tutorial_example1_network()
some_gen_id = n.get_generators().iloc[0].name
some_2wt_id = n.get_2_windings_transformers().iloc[0].name
some_shunt_id = n.get_shunt_compensators().iloc[0].name
params.add_constant_q_generators([some_gen_id])
params.add_variable_two_windings_transformers([some_2wt_id])
params.add_variable_shunt_compensators([some_shunt_id])
```

- Override the network voltage limits:

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
params.add_specific_low_voltage_limits([("vl_id_1", False, 380)])
params.add_specific_high_voltage_limits([("vl_id_2", False, 420)])
```

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
params.add_specificvoltage_limits({"vl_id": (0.5, 1.2)})
```

- Specify the objective function and the objective distance

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
params = v_init.VoltageInitializerParameters()
params.set_objective(va.VoltageInitializerObjective.SPECIFIC_VOLTAGE_PROFILE)
params.set_objective_distance(1.3)
```

- Tune scaling factors applied before ACOPF solving:

```
import pypowsybl as pp
import pypowsybl.voltage_initializer as v_init
```

(continues on next page)

(continued from previous page)

```
params = v_init.VoltageInitializerParameters()
params.set_default_variable_scaling_factor(1.1)
params.set_default_constraint_scaling_factor(0.9)
params.set_reactive_slack_variable_scaling_factor(0.15)
params.set_twt_ratio_variable_scaling_factor(0.002)
```

2.1.14 Advanced parameters

Part of pypowsybl is based on Java code compiled to a native library using GraalVM native image compiler. Compiled Java code relies on a small runtime (called SubstratVM) responsible for managing memory (this is essentially a garbage collector). Additional parameters can be passed to SubstratVM using the environment variable GRAALVM_OPTIONS.

By default, the maximum allowed memory (Xmx) is automatically defined based on machine memory. We can pass a specific value like this:

```
GRAALVM_OPTIONS="-Xmx1G" python
```

```
import logging
logging.basicConfig()
logging.getLogger('powsybl').setLevel(logging.DEBUG)
import pypowsybl as pp
DEBUG:powsybl:Max heap is 1086 MB
```

CHAPTER

THREE

API REFERENCE

For detailed description of pypowsybl classes and methods, please check out the API reference documentation :

3.1 API reference

This is the reference documentation for pypowsybl API. It provides a detailed description of pypowsybl classes and methods.

3.1.1 Network

The Network class is the representation of a power network in pypowsybl, it provides methods to access and modify underlying network elements data.

class Network(*handle*)

Parameters
handle (*_pp.JavaHandle*)

Network creation

Following methods may be used to create a new network instance:

is_loadable	Check if a file is a loadable network.
load	Load a network from a file.
load_from_string	Load a network from a string.
load_from_binary_buffer	Load a network from a binary buffer.
load_from_binary_buffers	Load a network from a list of binary buffers.
create_empty	Create an empty network.
create_ieee9	Create an instance of IEEE 9 bus network
create_ieee14	Create an instance of IEEE 14 bus network
create_ieee30	Create an instance of IEEE 30 bus network
create_ieee57	Create an instance of IEEE 57 bus network
create_ieee118	Create an instance of IEEE 118 bus network
create_ieee300	Create an instance of IEEE 300 bus network
create_eurostag_tutorial_example1_network	Create an instance of example 1 network of Eurostag tutorial
*create_eurostag_tutorial_example1_with_more_*	Create an instance of example 1 network of Eurostag tutorial, with a second generator
create_eurostag_tutorial_example1_with_powe	Create an instance of example 1 network of Eurostag tutorial with Power limits

continues on next page

Table 1 – continued from previous page

create_eurostag_tutorial_example1_with_tie_l	Create an instance of example 1 network of Eurostag tutorial with tie lines and areas
create_four_substations_node_breaker_networ	Create an instance of powsybl "4 substations" test case.
create_four_substations_node_breaker_networ	Create an instance of powsybl "4 substations" test case with ConnectablePosition and BusbarSectionPosition extensions.
create_micro_grid_be_network	Create an instance of micro grid BE CGMES test case
create_micro_grid_nl_network	Create an instance of micro grid NL CGMES test case
create_metrix_tutorial_six_buses_network	Create an instance of metrix tutorial six buses test case

pypowsybl.network.is_loadable

is_loadable(*file*)

Check if a file is a loadable network.

A file is loadable if:

- file exists
- it is a network format and format version supported by powsybl (so an importer exists for it)
- the network file is well formatted
- it is either not compressed or compressed with a supported compression format (gzip, bzip2, zip, xz)

Parameters
file (*str | PathLike*) – path to the supposed network file

Returns
True is the file is a loadable network, False otherwise

Return type
bool

pypowsybl.network.load

load(*file*, *parameters=None*, *post_processors=None*, *reporter=None*, *report_node=None*, *allow_variant_multi_thread_access=False*)

Load a network from a file. File should be in a supported format.

Basic compression formats are also supported (gzip, bzip2).

Parameters

- **file** (*str | PathLike*) – path to the network file
- **parameters** (*Dict[str, str] | None*) – a dictionary of import parameters
- **post_processors** (*List[str] | None*) – a list of import post processors (will be added to the ones defined by the platform config)
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)
- **allow_variant_multi_thread_access** (*bool*) – allow multi-thread access to variant (default: False)

Returns
The loaded network

Return type
Network

Examples

Some examples of file loading, including relative or absolute paths, and compressed files:

```
network = pp.network.load('network.xiidm')
network = pp.network.load('/path/to/network.xiidm')
network = pp.network.load('network.xiidm.gz')
network = pp.network.load('network.uct')
...
```

pypowsybl.network.load_from_string

load_from_string(*file_name*, *file_content*, *parameters=None*, *post_processors=None*, *reporter=None*, *report_node=None*, *allow_variant_multi_thread_access=False*)

Load a network from a string. File content should be in a supported format.

Parameters

- **file_name** (*str*) – file name
- **file_content** (*str*) – file content
- **parameters** (*Dict[str, str] | None*) – a dictionary of import parameters
- **post_processors** (*List[str] | None*) – a list of import post processors (will be added to the ones defined by the platform config)
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)
- **allow_variant_multi_thread_access** (*bool*) – allow multi-thread access to variant (default: False)

Returns
The loaded network

Return type
Network

pypowsybl.network.load_from_binary_buffer

load_from_binary_buffer(*buffer*, *parameters=None*, *post_processors=None*, *reporter=None*, *report_node=None*, *allow_variant_multi_thread_access=False*)

Load a network from a binary buffer.

Parameters

- **buffer** (*BytesIO*) – The BytesIO data buffer
- **parameters** (*Dict[str, str] | None*) – A dictionary of import parameters
- **post_processors** (*List[str] | None*) – a list of import post processors (will be added to the ones defined by the platform config)

- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)
- **allow_variant_multi_thread_access** (*bool*) – allow multi-thread access to variant (default: False)

Returns
The loaded network

Return type
Network

pypowsybl.network.load_from_binary_buffers

load_from_binary_buffers(*buffers*, *parameters=None*, *post_processors=None*, *reporter=None*, *report_node=None*, *allow_variant_multi_thread_access=False*)

Load a network from a list of binary buffers. Only zipped CGMES are supported for several zipped source load.

Parameters

- **buffers** (*List[BytesIO]*) – The list of BytesIO data buffer
- **parameters** (*Dict[str, str] | None*) – A dictionary of import parameters
- **post_processors** (*List[str] | None*) – a list of import post processors (will be added to the ones defined by the platform config)
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)
- **allow_variant_multi_thread_access** (*bool*) – allow multi-thread access to variant (default: False)

Returns
The loaded network

Return type
Network

pypowsybl.network.create_empty

create_empty(*network_id='Default'*, *allow_variant_multi_thread_access=False*)

Create an empty network.

Parameters

- **network_id** (*str*) – id of the network, defaults to 'Default'
- **allow_variant_multi_thread_access** (*bool*)

Returns
a new empty network

Return type
Network

pypowsybl.network.create_ieee9

create_ieee9(*allow_variant_multi_thread_access=False*)

Create an instance of IEEE 9 bus network

Returns
a new instance of IEEE 9 bus network

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_ieee14

create_ieee14(*allow_variant_multi_thread_access=False*)

Create an instance of IEEE 14 bus network

Returns
a new instance of IEEE 14 bus network

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_ieee30

create_ieee30(*allow_variant_multi_thread_access=False*)

Create an instance of IEEE 30 bus network

Returns
a new instance of IEEE 30 bus network

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_ieee57

create_ieee57(*allow_variant_multi_thread_access=False*)

Create an instance of IEEE 57 bus network

Returns
a new instance of IEEE 57 bus network

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_ieee118

create_ieee118(*allow_variant_multi_thread_access=False*)

Create an instance of IEEE 118 bus network

Returns
a new instance of IEEE 118 bus network

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_ieee300

create_ieee300(*allow_variant_multi_thread_access=False*)

Create an instance of IEEE 300 bus network

Returns
a new instance of IEEE 300 bus network

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_eurostag_tutorial_example1_network

create_eurostag_tutorial_example1_network(*allow_variant_multi_thread_access=False*)

Create an instance of example 1 network of Eurostag tutorial

Returns
a new instance of example 1 network of Eurostag tutorial

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_eurostag_tutorial_example1_with_more_generators_network

create_eurostag_tutorial_example1_with_more_generators_network(*allow_variant_multi_thread_access=False*)

Create an instance of example 1 network of Eurostag tutorial, with a second generator

Returns
a new instance of example 1 network of Eurostag tutorial with a second generator

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_eurostag_tutorial_example1_with_power_limits_network

create_eurostag_tutorial_example1_with_power_limits_network(*allow_variant_multi_thread_access=False*)

Create an instance of example 1 network of Eurostag tutorial with Power limits

Returns
a new instance of example 1 network of Eurostag tutorial with Power limits

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_eurostag_tutorial_example1_with_tie_lines_and_areas

create_eurostag_tutorial_example1_with_tie_lines_and_areas(*allow_variant_multi_thread_access=False*)

Create an instance of example 1 network of Eurostag tutorial with tie lines and areas

Returns
a new instance of example 1 network of Eurostag tutorial with tie lines and areas

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_four_substations_node_breaker_network

create_four_substations_node_breaker_network(*allow_variant_multi_thread_access=False*)

Create an instance of powsybl "4 substations" test case.

It is meant to contain most network element types that can be represented in powsybl networks. The topology is in node-breaker representation.

Returns
a new instance of powsybl "4 substations" test case

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_four_substations_node_breaker_network_with_extensions

create_four_substations_node_breaker_network_with_extensions(*allow_variant_multi_thread_access=False*)

Create an instance of powsybl "4 substations" test case with ConnectablePosition and BusbarSectionPosition extensions.

The topology is in node-breaker representation.

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_micro_grid_be_network

create_micro_grid_be_network(*allow_variant_multi_thread_access=False*)

Create an instance of micro grid BE CGMES test case

Returns
a new instance of micro grid BE CGMES test case

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_micro_grid_nl_network

create_micro_grid_nl_network(*allow_variant_multi_thread_access=False*)

Create an instance of micro grid NL CGMES test case

Returns
a new instance of micro grid NL CGMES test case

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

pypowsybl.network.create_metrix_tutorial_six_buses_network

create_metrix_tutorial_six_buses_network(*allow_variant_multi_thread_access=False*)

Create an instance of metrix tutorial six buses test case

Returns
a new instance of metrix tutorial six buses test case

Parameters
allow_variant_multi_thread_access (*bool*)

Return type
Network

Network properties

Network.id	ID of this network
Network.name	Name of this network
Network.source_format	Format of the source where this network came from.
Network.case_date	Date of this network case, in UTC timezone.
Network.forecast_distance	0 for a snapshot.
Network.per_unit	Defines if the network data should be used in per-unit.
Network.nominal_apparent_power	The nominal power to per unit the network (kVA)

pypowsybl.network.Network.id

property Network.id: str

ID of this network

pypowsybl.network.Network.name

property Network.name: str

Name of this network

pypowsybl.network.Network.source_format

property Network.source_format: str

Format of the source where this network came from.

pypowsybl.network.Network.case_date

property Network.case_date: datetime

Date of this network case, in UTC timezone.

pypowsybl.network.Network.forecast_distance

property Network.forecast_distance: timedelta

0 for a snapshot.

Type
The forecast distance

pypowsybl.network.Network.per_unit

property Network.per_unit: bool

Defines if the network data should be used in per-unit.

pypowsybl.network.Network.nominal_apparent_power

property Network.nominal_apparent_power: float

The nominal power to per unit the network (kVA)

Network elements access

All network elements are accessible as dataframes, using the following getters.

Note

Once obtained, a dataframe has no more relation to the network it originated from. In particular, changing a dataframe will not change the underlying network. Also, in order to get up-to-date data, for example after a loadflow execution, you will need to call again the corresponding getter.

Network.get_2_windings_transformers	Get a dataframe of 2 windings transformers.
Network.get_3_windings_transformers	Get a dataframe of 3 windings transformers.
Network.get_aliases	Get a dataframe of aliases of all network elements.
Network.get_areas	Get a dataframe of areas.
Network.get_areas_boundaries	Get a dataframe of areas boundaries.
Network.get_areas_voltage_levels	Get a dataframe of areas voltage levels.
Network.get_batteries	Get a dataframe of batteries.
Network.get_branches	Get a dataframe of branches
Network.get_busbar_sections	Get a dataframe of busbar sections.
Network.get_buses	Get a dataframe of buses from the bus view.
Network.get_bus_breaker_view_buses	Get a dataframe of buses from the bus/breaker view.
Network.get_boundary_lines	Get a dataframe of boundary lines.
Network.get_dangling_lines	
Network.get_boundary_lines_generation	Get a dataframe of boundary lines generation part.
Network.get_dangling_lines_generation	
Network.get_generators	Get a dataframe of generators.
Network.get_hvdc_lines	Get a dataframe of HVDC lines.
Network.get_identifiables	Get a dataframe of identifiables
Network.get_injections	Get a dataframe of injections
Network.get_lcc_converter_stations	Get a dataframe of LCC converter stations.
Network.get_lines	Get a dataframe of lines data.
Network.get_loads	Get a dataframe of loads.
Network.get_linear_shunt_compensator_sectio	Get a dataframe of shunt compensators sections for linear model.
Network.get_non_linear_shunt_compensator_se	Get a dataframe of shunt compensators sections for non linear model.
Network.get_operational_limits	Get the list of operational limits.
Network.get_phase_tap_changer_steps	Get a dataframe of phase tap changer steps.
Network.get_phase_tap_changers	Create a phase tap changers:class:*~pandas.DataFrame*.
Network.get_ratio_tap_changer_steps	Get a dataframe of ratio tap changer steps.
Network.get_ratio_tap_changers	Create a ratio tap changers:class:*~pandas.DataFrame*.
Network.get_reactive_capability_curve_points	Get a dataframe of reactive capability curve points.
Network.get_shunt_compensators	Get a dataframe of shunt compensators.
Network.get_static_var_compensators	Get a dataframe of static var compensators.
Network.get_substations	Get substations `DataFrame`.
Network.get_switches	Get a dataframe of switches.
Network.get_terminals	Get a dataframe of terminal
Network.get_voltage_levels	Get a dataframe of voltage levels.
Network.get_vsc_converter_stations	Get a dataframe of VSC converter stations.
Network.get_tie_lines	Get a dataframe of tie lines.
Network.get_grounds	Get a dataframe of grounds.
Network.get_sub_networks	Get a dataframe of sub networks
Network.get_sub_network	Get a sub network from its parent network.
Network.detach	Detach a sub network from its parent network.
Network.get_dc_nodes	Get a dataframe of dc nodes.
Network.get_dc_lines	Get a dataframe of DC lines
Network.get_voltage_source_converters	Get a dataframe of voltage source converters.
Network.get_dc_grounds	Get a dataframe of dc grounds.
Network.get_dc_buses	Get a dataframe of dc buses.

pypowsybl.network.Network.get_2_windings_transformers

Network.**get_2_windings_transformers**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of 2 windings transformers.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of 2 windings transformers.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **r**: the resistance of the transformer at its "2" side (in Ohm)
- **x**: the reactance of the transformer at its "2" side (in Ohm)
- **b**: the susceptance of transformer at its "2" side (in Siemens)
- **g**: the conductance of transformer at its "2" side (in Siemens)
- **rated_u1**: the rated voltage of the transformer at side 1 (in kV)
- **rated_u2**: the rated voltage of the transformer at side 2 (in kV)
- **rated_s**: the rated apparent power of the transformer (in MVA)
- **p1**: the active flow on the transformer at its "1" side, `NaN` if no loadflow has been computed (in MW)
- **q1**: the reactive flow on the transformer at its "1" side, `NaN` if no loadflow has been computed (in MVAr)
- **i1**: the current on the transformer at its "1" side, `NaN` if no loadflow has been computed (in A)
- **p2**: the active flow on the transformer at its "2" side, `NaN` if no loadflow has been computed (in MW)
- **q2**: the reactive flow on the transformer at its "2" side, `NaN` if no loadflow has been computed (in MVAr)
- **i2**: the current on the transformer at its "2" side, `NaN` if no loadflow has been computed (in A)
- **voltage_level1_id**: voltage level where the transformer is connected, on side 1
- **voltage_level2_id**: voltage level where the transformer is connected, on side 2
- **bus1_id**: bus where this transformer is connected, on side 1
- **bus2_id**: bus where this transformer is connected, on side 2
- **bus_breaker_bus1_id** (optional): bus of the bus-breaker view where this transformer is connected, on side 1
- **bus_breaker_bus2_id** (optional): bus of the bus-breaker view where this transformer is connected, on side 2

- **node1** (optional): node where this transformer is connected on side 1, in node-breaker voltage levels
- **node2** (optional): node where this transformer is connected on side 2, in node-breaker voltage levels
- **connected1**: `True` if the side "1" of the transformer is connected to a bus
- **connected2**: `True` if the side "2" of the transformer is connected to a bus
- **fictitious** (optional): `True` if the transformer is part of the model and not of the actual network
- **selected_limits_group_1** (optional): Name of the selected operational limits group selected for side 1
- **selected_limits_group_2** (optional): Name of the selected operational limits group selected for side 2
- **rho** (optional): the voltage ratio of the transformer at current tap position
- **alpha** (optional): the phase shift of the transformer at current tap position (in degree)
- **r_at_current_tap** (optional): the resistance of the transformer at current tap position (in Ohm)
- **x_at_current_tap** (optional): the reactance of the transformer at current tap position (in Ohm)
- **g_at_current_tap** (optional): the susceptance of the transformer at current tap position (in Ohm)
- **b_at_current_tap** (optional): the conductance of the transformer at current tap position (in Ohm)

This dataframe is indexed by the id of the two windings transformers

Examples

```
net = pp.network.create_ieee14()
net.get_2_windings_transformers()
```

will output something like:

	r	x	g	b	rate	rate	rate	p1	q1	i1	p2	q2	i2	volt age	volt age	bus	bus	con nec	con-nected2
id																			
T4-7-1	0.0	0.4(	0.0	0.0	132.	14.(	NaN	NaN	NaN	NaN	NaN	NaN	NaN	VL4	VL7	VL4	VL7	True	True
T4-9-1	0.0	0.8(	0.0	0.0	130.	12.(	NaN	NaN	NaN	NaN	NaN	NaN	NaN	VL4	VL9	VL4	VL9	True	True
T5-6-1	0.0	0.3(	0.0	0.0	125.	12.(	NaN	NaN	NaN	NaN	NaN	NaN	NaN	VL5	VL(	VL5	VL(	True	True

```
net = pp.network.create_ieee14()
net.get_2_windings_transformers(all_attributes=True)
```

will output something like:

	r	x	g	b	rate	rate	rate	p1	q1	i1	p2	q2	i2	volt age	volt age	bus	bus	con nec	con-nected2
id																			
T4-7-1	0.0	0.4(	0.0	0.0	132.	14.(	NaN	NaN	NaN	NaN	NaN	NaN	NaN	VL4	VL7	VL4	VL7	True	True
T4-9-1	0.0	0.8(	0.0	0.0	130.	12.(	NaN	NaN	NaN	NaN	NaN	NaN	NaN	VL4	VL9	VL4	VL9	True	True
T5-6-1	0.0	0.3(	0.0	0.0	125.	12.(	NaN	NaN	NaN	NaN	NaN	NaN	NaN	VL5	VL6	VL5	VL6	True	True

```
net = pp.network.create_ieee14()
net.get_2_windings_transformers(attributes=['p1','q1','i1','p2','q2','i2','voltage_
↪level1_id','voltage_level2_id','bus1_id','bus2_id','connected1','connected2'])
```

will output something like:

	p1	q1	i1	p2	q2	i2	volt-age_level1_	volt-age_level2_	bus1_	bus2_	con-nected1	con-nected2
id												
T4-7-1	NaN	NaN	NaN	NaN	NaN	NaN	VL4	VL7	VL4_(	VL7_(	True	True
T4-9-1	NaN	NaN	NaN	NaN	NaN	NaN	VL4	VL9	VL4_(	VL9_(	True	True
T5-6-1	NaN	NaN	NaN	NaN	NaN	NaN	VL5	VL6	VL5_(	VL6_(	True	True

pypowsybl.network.Network.get_3_windings_transformers

Network.**get_3_windings_transformers**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of 3 windings transformers.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of 3 windings transformers.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **rated_u0**: the rated voltage of the transformer at middle point od the star model (in kV)
- **fictitious** (optional): `True` if the transformer is part of the model and not of the actual network
- **r1**: the leg 1 resistance of the transformer (in Ohm)
- **x1**: the leg 1 reactance of the transformer (in Ohm)
- **b1**: the leg 1 susceptance of transformer (in Siemens)
- **g1**: the leg 1 conductance of transformer (in Siemens)
- **rated_u1**: the leg 1 rated voltage of the transformer (in kV)
- **rated_s1**: the leg 1 rated apparent power of the transformer (in MVA)
- **ratio_tap_position1**: the leg 1 ratio tap changer current position
- **phase_tap_position1**: the leg 1 phase tap changer current position
- **p1**: the leg 1 active power flow on the transformer, `NaN` if no loadflow has been computed (in MW)
- **q1**: the leg 1 reactive power flow on the transformer, `NaN` if no loadflow has been computed (in MVAr)
- **i1**: the leg 1 current on the transformer, `NaN` if no loadflow has been computed (in A)
- **voltage_level1_id**: the voltage level where the leg 1 of the transformer is connected
- **bus1_id**: the bus where the leg 1 of the transformer is connected
- **bus_breaker_bus1_id** (optional): the bus of the bus-breaker view where leg 1 of the transformer is connected
- **node1** (optional): the node where the leg 1 transformer is connected (only in node-breaker voltage levels)
- **connected1**: `True` if the leg 1 of the transformer is connected to a bus
- **selected_limits_group_1** (optional): the name of the selected operational limits group selected for the leg 1 of the transformer
- **rho1** (optional): the leg 1 voltage ratio of the transformer at current tap position
- **alpha1** (optional): the leg 1 phase shift of the transformer at current tap position (in degree)
- **r1_at_current_tap** (optional): the leg 1 resistance of the transformer at current tap position (in Ohm)
- **x1_at_current_tap** (optional): the leg 1 reactance of the transformer at current tap position (in Ohm)
- **g1_at_current_tap** (optional): the leg 1 susceptance of the transformer at current tap position (in Ohm)
- **b1_at_current_tap** (optional): the leg 1 conductance of the transformer at current tap position (in Ohm)
- **r2**: the leg 2 resistance of the transformer (in Ohm)
- **x2**: the leg 2 reactance of the transformer (in Ohm)
- **b2**: the leg 2 susceptance of transformer (in Siemens)
- **g2**: the leg 2 conductance of transformer (in Siemens)
- **rated_u2**: the leg 2 rated voltage of the transformer (in kV)
- **rated_s2**: the leg 2 rated apparent power of the transformer (in MVA)
- **ratio_tap_position2**: the leg 2 ratio tap changer current position
- **phase_tap_position2**: the leg 2 phase tap changer current position

- **p2**: the leg 2 active power flow on the transformer, `NaN` if no loadflow has been computed (in MW)
- **q2**: the leg 2 reactive power flow on the transformer, `NaN` if no loadflow has been computed (in MVAr)
- **i2**: the leg 2 current on the transformer, `NaN` if no loadflow has been computed (in A)
- **voltage_level2_id**: the voltage level where the leg 2 of the transformer is connected
- **bus2_id**: the bus where the leg 2 of the transformer is connected
- **bus_breaker_bus2_id** (optional): the bus of the bus-breaker view where leg 2 of the transformer is connected
- **node2** (optional): the node where the leg 2 transformer is connected (only in node-breaker voltage levels)
- **connected2**: `True` if the leg 2 of the transformer is connected to a bus
- **selected_limits_group_2** (optional): the name of the selected operational limits group selected for the leg 2 of the transformer
- **rho2** (optional): the leg 2 voltage ratio of the transformer at current tap position
- **alpha2** (optional): the leg 2 phase shift of the transformer at current tap position (in degree)
- **r2_at_current_tap** (optional): the leg 2 resistance of the transformer at current tap position (in Ohm)
- **x2_at_current_tap** (optional): the leg 2 reactance of the transformer at current tap position (in Ohm)
- **g2_at_current_tap** (optional): the leg 2 susceptance of the transformer at current tap position (in Ohm)
- **b2_at_current_tap** (optional): the leg 2 conductance of the transformer at current tap position (in Ohm)
- **r3**: the leg 3 resistance of the transformer (in Ohm)
- **x3**: the leg 3 reactance of the transformer (in Ohm)
- **b3**: the leg 3 susceptance of transformer (in Siemens)
- **g3**: the leg 3 conductance of transformer (in Siemens)
- **rated_u3**: the leg 3 rated voltage of the transformer (in kV)
- **rated_s3**: the leg 3 rated apparent power of the transformer (in MVA)
- **ratio_tap_position3**: the leg 3 ratio tap changer current position
- **phase_tap_position3**: the leg 3 phase tap changer current position
- **p3**: the leg 3 active power flow on the transformer, `NaN` if no loadflow has been computed (in MW)
- **q3**: the leg 3 reactive power flow on the transformer, `NaN` if no loadflow has been computed (in MVAr)
- **i3**: the leg 3 current on the transformer, `NaN` if no loadflow has been computed (in A)
- **voltage_level3_id**: the voltage level where the leg 3 of the transformer is connected
- **bus3_id**: the bus where the leg 3 of the transformer is connected
- **bus_breaker_bus3_id** (optional): the bus of the bus-breaker view where leg 3 of the transformer is connected
- **node3** (optional): the node where the leg 3 transformer is connected (only in node-breaker voltage levels)
- **connected3**: `True` if the leg 3 of the transformer is connected to a bus
- **selected_limits_group_3** (optional): the name of the selected operational limits group selected for the leg 3 of the transformer
- **rho3** (optional): the leg 3 voltage ratio of the transformer at current tap position

- **alpha3** (optional): the leg 3 phase shift of the transformer at current tap position (in degree)
- **r3_at_current_tap** (optional): the leg 3 resistance of the transformer at current tap position (in Ohm)
- **x3_at_current_tap** (optional): the leg 3 reactance of the transformer at current tap position (in Ohm)
- **g3_at_current_tap** (optional): the leg 3 susceptance of the transformer at current tap position (in Ohm)
- **b3_at_current_tap** (optional): the leg 3 conductance of the transformer at current tap position (in Ohm)

This dataframe is indexed by the id of the three windings transformers

pypowsybl.network.Network.get_aliases

Network.**get_aliases**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of aliases of all network elements.

Args:

Returns
A dataframe of aliases

Parameters

- **all_attributes** (*bool*)
- **attributes** (*List[str] | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **type**: the type of the network element (network, line, generator, load, ...)
- **alias**: alias value
- **alias_type**: alias type

This dataframe is indexed on the network element ID.

pypowsybl.network.Network.get_areas

Network.**get_areas**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of areas.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
the areas dataframe

Return type
DataFrame

See also

- *get_areas_voltage_levels()* to retrieve the voltage levels of the areas
- *get_areas_boundaries()* to retrieve the voltage levels of the areas boundaries
- *create_areas()* to create areas
- *update_areas()* to update areas

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **area_type**: the type of area (e.g. ControlArea, BiddingZone, ...)
- **interchange_target**: target active power interchange (MW)
- **interchange**: total (AC + DC) active power interchange, in load sign convention (negative is export, positive is import) (MW)
- **ac_interchange**: AC active power interchange, in load sign convention (negative is export, positive is import) (MW)
- **dc_interchange**: DC active power interchange, in load sign convention (negative is export, positive is import) (MW)
- **fictitious** (optional): `True` if the area is part of the model and not of the actual network

This dataframe is indexed on the area ID.

Examples

```
net = pp.network.create_eurostag_tutorial_example1_with_tie_lines_and_areas()
net.get_areas()
```

will output something like:

	name	area_type	inter-change_target	inter-change	ac_interchang	dc_interchange
id						
Contro-lArea_A	Control Area A	Contro-lArea	-602.6	-602.948693	-602.948693	0.0
Contro-lArea_B	Control Area B	Contro-lArea	602.6	602.944639	602.944639	0.0
Region_AB	Region AB	Region	NaN	0.000000	0.000000	0.0

pypowsybl.network.Network.get_areas_boundaries

Network.**get_areas_boundaries**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of areas boundaries.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
the areas boundaries dataframe

Return type
DataFrame

See also

create_areas_boundaries()

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **id**: area identifier
- **boundary_type** (optional): either *BOUNDARY_LINE* or *TERMINAL*
- **element**: either identifier of the boundary Line or the equipment terminal
- **side** (optional): equipment side
- **ac**: True if the boundary is considered as AC and not DC
- **p**: Active power at boundary (MW)
- **q**: Reactive power at boundary (MW)

This dataframe is indexed on the area ID.

Examples

```
net = pp.network.create_eurostag_tutorial_example1_with_tie_lines_and_areas()
net.get_areas_boundaries()
```

will output something like:

	element	ac	p	q
id				
ControlArea_A	NHV1_XNODE1	True	-301.474347	-116.518644
ControlArea_A	NVH1_XNODE2	True	-301.474347	-116.518644
ControlArea_B	XNODE1_NHV2	True	301.472320	116.434157
ControlArea_B	XNODE2_NHV2	True	301.472320	116.434157

pypowsybl.network.Network.get_areas_voltage_levels

Network.**get_areas_voltage_levels**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of areas voltage levels.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

the areas voltage levels dataframe

Return type

DataFrame

See also

create_areas_voltage_levels()

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **id**: area identifier
- **voltage_level_id**: voltage level identifier

This dataframe is indexed on the area ID.

Examples

```
net = pp.network.create_eurostag_tutorial_example1_with_tie_lines_and_areas()
net.get_areas_voltage_levels()
```

will output something like:

	voltage_level_id
id	
ControlArea_A	VLGEN
ControlArea_A	VLHV1
ControlArea_B	VLHV2
ControlArea_B	VLLOAD
Region_AB	VLGEN
Region_AB	VLHV1
Region_AB	VLHV2
Region_AB	VLLOAD

pypowsybl.network.Network.get_batteries

Network.**get_batteries**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of batteries.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of batteries.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **name**: type of load
- **max_p**: the maximum active value for the battery (MW)
- **min_p**: the minimum active value for the battery (MW)
- **min_q**: the maximum reactive value for the battery only if reactive_limits_kind is MIN_MAX (MVar)
- **max_q**: the minimum reactive value for the battery only if reactive_limits_kind is MIN_MAX (MVar)
- **max_q_at_target_p** (optional): the maximum reactive value for the battery for the target p specified (MVar)
- **min_q_at_target_p** (optional): the minimum reactive value for the battery for the target p specified (MVar)
- **max_q_at_p** (optional): the maximum reactive value for the battery at current p (MVar)
- **min_q_at_p** (optional): the minimum reactive value for the battery at current p (MVar)
- **target_p**: The active power setpoint (MW)
- **target_q**: The reactive power setpoint (MVAr)

- **p**: the result active battery consumption, it is NaN is not loadflow has been computed (MW)
- **q**: the result reactive battery consumption, it is NaN is not loadflow has been computed (MVAr)
- **i**: the current on the battery, NaN if no loadflow has been computed (in A)
- **voltage_level_id**: at which substation this load is connected
- **bus_id**: bus where this load is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this battery is connected
- **node** (optional): node where this battery is connected, in node-breaker voltage levels
- **connected**: True if the battery is connected to a bus
- **fictitious** (optional): True if the battery is part of the model and not of the actual network

This dataframe is indexed on the battery ID.

pypowsybl.network.Network.get_branches

Network.**get_branches**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of branches

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Returns
A dataframe of branches.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **type**: the type of the branch (line or 2 windings transformer)
- **voltage_level1_id**: voltage level where the branch is connected, on side 1
- **node1** (optional): node where this branch is connected on side 1, in node-breaker voltage levels
- **bus_breaker_bus1_id** (optional): bus of the bus-breaker view where this branch is connected, on side "1"
- **connected1**: True if the side "1" of the branch is connected to a bus
- **bus1_id**: bus where this branch is connected, on side 1
- **voltage_level2_id**: voltage level where the branch is connected, on side 2
- **node2** (optional): node where this branch is connected on side 2, in node-breaker voltage levels
- **bus_breaker_bus2_id** (optional): bus of the bus-breaker view where this branch is connected, on side "2"
- **connected2**: True if the side "2" of the branch is connected to a bus

- **bus2_id**: bus where this branch is connected, on side 2
- **p1**: the active flow on the branch at its "1" side, NaN if no loadflow has been computed (in MW)
- **q1**: the reactive flow on the branch at its "1" side, NaN if no loadflow has been computed (in MVAr)
- **i1**: the current on the branch at its "1" side, NaN if no loadflow has been computed (in A)
- **p2**: the active flow on the branch at its "2" side, NaN if no loadflow has been computed (in MW)
- **q2**: the reactive flow on the branch at its "2" side, NaN if no loadflow has been computed (in MVAr)
- **i2**: the current on the branch at its "2" side, NaN if no loadflow has been computed (in A)
- **selected_limits_group_1** (optional): Name of the selected operational limits group selected for side 1
- **selected_limits_group_2** (optional): Name of the selected operational limits group selected for side 2

This dataframe is indexed on the branch ID.

pypowsybl.network.Network.get_busbar_sections

Network.**get_busbar_sections**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of busbar sections.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of busbar sections.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **fictitious** (optional): True if the busbar section is part of the model and not of the actual network
- **v**: The voltage magnitude of the busbar section (in kV)
- **angle**: the voltage angle of the busbar section (in degree)
- **voltage_level_id**: at which substation the busbar section is connected
- **bus_id**: bus this busbar section belongs to
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view this busbar section belongs to
- **node** (optional): node associated to the this busbar section, in node-breaker voltage levels
- **connected**: True if the busbar section is connected to a bus

This dataframe is indexed by the id of the busbar sections

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_busbar_sections()
```

will output something like:

	name	v	angle	voltage_level_id	bus_id	connected
id						
S1VL1_BBS	S1VL1_BBS	224.6139	2.2822	S1VL1	S1VL1_0	True
S1VL2_BBS1	S1VL2_BBS1	400.0000	0.0000	S1VL2	S1VL2_0	True
S1VL2_BBS2	S1VL2_BBS2	400.0000	0.0000	S1VL2	S1VL2_0	True
S2VL1_BBS	S2VL1_BBS	408.8470	0.7347	S2VL1	S2VL1_0	True
S3VL1_BBS	S3VL1_BBS	400.0000	0.0000	S3VL1	S3VL1_0	True
S4VL1_BBS	S4VL1_BBS	400.0000	-1.1259	S4VL1	S4VL1_0	True

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_busbar_sections(all_attributes=True)
```

will output something like:

	name	v	angle	voltage_level_id	bus_id	bus_breaker_b	node	connected	fictitious
id									
S1VL1_BE	S1VL1_BE	224.613	2.2822	S1VL1	S1VL1_	S1VL1_0	0	True	False
S1VL2_BE	S1VL2_BE	400.000	0.0000	S1VL2	S1VL2_	S1VL2_0	0	True	False
S1VL2_BE	S1VL2_BE	400.000	0.0000	S1VL2	S1VL2_	S1VL2_1	1	True	False
S2VL1_BE	S2VL1_BE	408.847	0.7347	S2VL1	S2VL1_	S2VL1_0	0	True	False
S3VL1_BE	S3VL1_BE	400.000	0.0000	S3VL1	S3VL1_	S3VL1_0	0	True	False
S4VL1_BE	S4VL1_BE	400.000	-1.1259	S4VL1	S4VL1_	S4VL1_0	0	True	False

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_busbar_sections(attributes=['v','angle','voltage_level_id','connected'])
```

will output something like:

	v	angle	voltage_level_id	connected
id				
S1VL1_BBS	224.6139	2.2822	S1VL1	True
S1VL2_BBS1	400.0000	0.0000	S1VL2	True
S1VL2_BBS2	400.0000	0.0000	S1VL2	True
S2VL1_BBS	408.8470	0.7347	S2VL1	True
S3VL1_BBS	400.0000	0.0000	S3VL1	True
S4VL1_BBS	400.0000	-1.1259	S4VL1	True

pypowsybl.network.Network.get_buses

Network.**get_buses**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of buses from the bus view.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of buses from the bus view

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **v_mag**: Get the voltage magnitude of the bus (in kV)
- **v_angle**: the voltage angle of the bus (in degree)
- **connected_component**: The connected component to which the bus belongs
- **synchronous_component**: The synchronous component to which the bus belongs
- **voltage_level_id**: at which substation the bus is connected

This dataframe is indexed on the bus ID in the bus view.

See also

get_bus_breaker_view_buses()

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_buses()
```

It outputs something like:

	v_mag	v_angle	connected_component	synchronous_component	voltage_level_id
id					
S1VL1_0	224.6139	2.2822	0	1	S1VL1
S1VL2_0	400.0000	0.0000	0	1	S1VL2
S2VL1_0	408.8470	0.7347	0	0	S2VL1
S3VL1_0	400.0000	0.0000	0	0	S3VL1
S4VL1_0	400.0000	-1.1259	0	0	S4VL1

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_buses(all_attributes=True)
```

It outputs something like:

	v_mag	v_angle	connected_component	synchronous_component	voltage_level_id
id					
S1VL1_0	224.6139	2.2822	0	1	S1VL1
S1VL2_0	400.0000	0.0000	0	1	S1VL2
S2VL1_0	408.8470	0.7347	0	0	S2VL1
S3VL1_0	400.0000	0.0000	0	0	S3VL1
S4VL1_0	400.0000	-1.1259	0	0	S4VL1

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_buses(attributes=['v_mag','v_angle','voltage_level_id'])
```

It outputs something like:

	v_mag	v_angle	voltage_level_id
id			
S1VL1_0	224.6139	2.2822	S1VL1
S1VL2_0	400.0000	0.0000	S1VL2
S2VL1_0	408.8470	0.7347	S2VL1
S3VL1_0	400.0000	0.0000	S3VL1
S4VL1_0	400.0000	-1.1259	S4VL1

pypowsybl.network.Network.get_bus_breaker_view_buses

Network.**get_bus_breaker_view_buses**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of buses from the bus/breaker view.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of buses from the bus/breaker view

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **v_mag**: Get the voltage magnitude of the bus (in kV)
- **v_angle**: the voltage angle of the bus (in degree)
- **connected_component**: The connected component to which the bus belongs
- **synchronous_component**: The synchronous component to which the bus belongs
- **voltage_level_id**: at which substation the bus is connected
- **bus_id**: the bus ID in the bus view

This dataframe is indexed on the bus ID in the bus/breaker view.

See also

get_buses()

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_bus_breaker_view_buses()
```

It outputs something like:

	name	v_mag	v_angle	con-nected_cor	syn-chronous_c	volt-age_level_i	bus_id
id							
S1VL1_0		224.6139	2.2822	0	1	S1VL1	S1VL1_0
S1VL1_2		224.6139	2.2822	0	1	S1VL1	S1VL1_0

continues on next page

Table 4 – continued from previous page

	name	v_mag	v_angle	con-nected_cor	syn-chronous_c	volt-age_level_i	bus_id
S1VL1_4		224.6139	2.2822	0	1	S1VL1	S1VL1_0
S1VL2_0		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_1		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_3		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_5		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_7		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_9		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_11		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_13		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_15		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_17		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_19		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S1VL2_21		400.0000	0.0000	0	1	S1VL2	S1VL2_0
S2VL1_0		408.8470	0.7347	0	0	S2VL1	S2VL1_0
S2VL1_2		408.8470	0.7347	0	0	S2VL1	S2VL1_0
S2VL1_4		408.8470	0.7347	0	0	S2VL1	S2VL1_0
S2VL1_6		408.8470	0.7347	0	0	S2VL1	S2VL1_0
S3VL1_0		400.0000	0.0000	0	0	S3VL1	S3VL1_0
S3VL1_2		400.0000	0.0000	0	0	S3VL1	S3VL1_0
S3VL1_4		400.0000	0.0000	0	0	S3VL1	S3VL1_0
S3VL1_6		400.0000	0.0000	0	0	S3VL1	S3VL1_0
S3VL1_8		400.0000	0.0000	0	0	S3VL1	S3VL1_0
S3VL1_10		400.0000	0.0000	0	0	S3VL1	S3VL1_0
S4VL1_0		400.0000	-1.1259	0	0	S4VL1	S4VL1_0
S4VL1_6		400.0000	-1.1259	0	0	S4VL1	S4VL1_0
S4VL1_2		400.0000	-1.1259	0	0	S4VL1	S4VL1_0
S4VL1_4		400.0000	-1.1259	0	0	S4VL1	S4VL1_0

pypowsybl.network.Network.get_boundary_lines

`Network.`**`get_boundary_lines`**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of boundary lines.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of boundary lines.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **r**: The resistance of the boundary line (Ohm)
- **x**: The reactance of the boundary line (Ohm)
- **g**: the conductance of boundary line (in Siemens)
- **b**: the susceptance of boundary line (in Siemens)
- **p0**: The active power setpoint
- **q0**: The reactive power setpoint
- **p**: active flow on the boundary line, `NaN` if no loadflow has been computed (in MW)
- **q**: the reactive flow on the boundary line, `NaN` if no loadflow has been computed (in MVAr)
- **i**: The current on the boundary line, `NaN` if no loadflow has been computed (in A)
- **boundary_p** (optional): active flow on the boundary line at boundary bus side, `NaN` if no loadflow has been computed (in MW)
- **boundary_q** (optional): reactive flow on the boundary line at boundary bus side, `NaN` if no loadflow has been computed (in MW)
- **boundary_i** (optional): current on the boundary line at boundary bus side, `NaN` if no loadflow has been computed (in A)
- **boundary_v_mag** (optional): voltage magnitude of the boundary bus, `NaN` if no loadflow has been computed (in kV)
- **boundary_v_angle** (optional): voltage angle of the boundary bus, `NaN` if no loadflow has been computed (in degree)
- **voltage_level_id**: at which substation the boundary line is connected
- **bus_id**: bus where this line is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this line is connected
- **node** (optional): node where this line is connected, in node-breaker voltage levels
- **connected**: `True` if the boundary line is connected to a bus
- **fictitious** (optional): `True` if the boundary line is part of the model and not of the actual network
- **pairing_key**: the pairing key associated to the boundary line, to be used for creating tie lines.
- **ucte_xnode_code**: deprecated for pairing_key.
- **paired**: if the boundary line is paired with a tie line
- **tie_line_id**: the ID of the tie line if the boundary line is paired

This dataframe is indexed by the id of the boundary lines

Examples

```
net = pp.network.create_boundary_lines_network()
net.get_boundary_lines()
```

will output something like:

	r	x	g	b	p0	q0	p	q	i	volt-age_level_id	bus_id	con-nected
id												
BL	10.0	1.0	0.0001	0.00001	50.0	30.0	NaN	NaN	NaN	VL	VL_0	True

```
net = pp.network.create_boundary_lines_network()
net.get_boundary_lines(all_attributes=True)
```

will output something like:

	r	x	g	b	p0	q0	p	q	i	volt-age_level_id	bus_id	con-nected
id												
BL	10.0	1.0	0.0001	0.00001	50.0	30.0	NaN	NaN	NaN	VL	VL_0	True

```
net = pp.network.create_boundary_lines_network()
net.get_boundary_lines(attributes=['p','q','i','voltage_level_id','bus_id',
↪'connected'])
```

will output something like:

	p	q	i	voltage_level_id	bus_id	connected
id						
BL	NaN	NaN	NaN	VL	VL_0	True

Note

This note applies only if you are using the per-unit mode in your network (i.e., network.per_unit=True).

If two boundary lines are paired in a tie-line and have different nominal voltages, the per-unit values for *boundary_i* and *boundary_v_mag* will differ between the two boundary lines.

Currently, PowSyBl network model does not support the concept of nominal voltage for the boundary fictitious bus. Therefore, the nominal voltage at the boundary line network side is used for per-unit calculations. While this is generally not an issue, this produces counterintuitive results in the case of boundary lines of different nominal voltages.

pypowsybl.network.Network.get_dangling_lines

Network.**get_dangling_lines**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Deprecated since version 1.15.0: Use *get_boundary_lines()* instead.

Parameters

- **all_attributes** (*bool*)
- **attributes** (*List[str] | None*)

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
DataFrame

pypowsybl.network.Network.get_boundary_lines_generation

Network.**get_boundary_lines_generation**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of boundary lines generation part.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of boundary lines generation part.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **min_p**: Minimum active power output of the boundary line's generation part
- **max_p**: Maximum active power output of the boundary line's generation part
- **target_p**: Active power target of the generation part
- **target_q**: Reactive power target of the generation part
- **target_v**: Voltage target of the generation part
- **voltage_regulator_on**: `True` if the generation part regulates voltage

This dataframe is indexed by the id of the boundary lines.

pypowsybl.network.Network.get_dangling_lines_generation

Network.**get_dangling_lines_generation**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Deprecated since version 1.15.0: Use *get_boundary_lines_generation()* instead.

Parameters

- **all_attributes** (*bool*)
- **attributes** (*List[str] | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
DataFrame

pypowsybl.network.Network.get_generators

Network.**get_generators**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of generators.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
the generators dataframe.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **energy_source**: the energy source used to fuel the generator
- **target_p**: the target active value for the generator (in MW)
- **max_p**: the maximum active value for the generator (MW)
- **min_p**: the minimum active value for the generator (MW)
- **max_q**: the maximum reactive value for the generator only if reactive_limits_kind is MIN_MAX (MVar)
- **min_q**: the minimum reactive value for the generator only if reactive_limits_kind is MIN_MAX (MVar)
- **max_q_at_target_p** (optional): the maximum reactive value for the generator for the target p specified (MVar)
- **min_q_at_target_p** (optional): the minimum reactive value for the generator for the target p specified (MVar)
- **max_q_at_p** (optional): the maximum reactive value for the generator at current p (MVar)
- **min_q_at_p** (optional): the minimum reactive value for the generator at current p (MVar)
- **rated_s**: The rated nominal power (MVA)
- **reactive_limits_kind**: type of the reactive limit of the generator (can be MIN_MAX, CURVE or NONE)
- **target_v**: the target voltage magnitude value for the generator (in kV)
- **equivalent_local_target_v (optional): a local target voltage value expected to be consistent with the remote target voltage (in kV)**
  (to be used by simulators that deactivate the remote voltage algorithms, or by dynamic simulators that use this voltage as a starting value)
- **target_q**: the target reactive value for the generator (in MVAr)

- **voltage_regulator_on**: `True` if the generator regulates voltage
- **regulated_element_id**: the ID of the network element where voltage is regulated
- **regulated_bus_id**: the ID of the bus from bus view where voltage is regulated
- **regulated_bus_breaker_bus_id**: the ID of the bus from bus/breaker view where voltage is regulated
- **p**: the actual active production of the generator (`NaN` if no loadflow has been computed)
- **q**: the actual reactive production of the generator (`NaN` if no loadflow has been computed)
- **i**: the current on the generator, `NaN` if no loadflow has been computed (in A)
- **voltage_level_id**: at which substation this generator is connected
- **bus_id**: bus where this generator is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this generator is connected
- **node** (optional): node where this generator is connected, in node-breaker voltage levels
- **condenser** (optional): `True` if the generator is a condenser
- **connected**: `True` if the generator is connected to a bus
- **fictitious** (optional): `True` if the generator is part of the model and not of the actual network

This dataframe is indexed on the generator ID.

Examples

```
net = pp.network.create_ieee14()
net.get_generators()
```

will output something like:

	en-ergy_sour	tar-get_p	max_	min_	tar-get_v	tar-get_q	volt-age_regulator	p	q	volt-age_level_	bus_id
id											
B1-G	OTHER	232.4	9999.(	-9999.(	1.060	-16.9	True	NaN	NaN	VL1	VL1_0
B2-G	OTHER	40.0	9999.(	-9999.(	1.045	42.4	True	NaN	NaN	VL2	VL2_0
B3-G	OTHER	0.0	9999.(	-9999.(	1.010	23.4	True	NaN	NaN	VL3	VL3_0
B6-G	OTHER	0.0	9999.(	-9999.(	1.070	12.2	True	NaN	NaN	VL6	VL6_0
B8-G	OTHER	0.0	9999.(	-9999.(	1.090	17.4	True	NaN	NaN	VL8	VL8_0

```
net = pp.network.create_ieee14()
net.get_generators(all_attributes=True)
```

will output something like:

	en-ergy_sour	tar-get_p	max_	min_	tar-get_v	tar-get_q	volt-age_regulator	p	q	volt-age_level_	bus_id
id											
B1-G	OTHER	232.4	9999.(	-9999.(	1.060	-16.9	True	NaN	NaN	VL1	VL1_0
B2-G	OTHER	40.0	9999.(	-9999.(	1.045	42.4	True	NaN	NaN	VL2	VL2_0
B3-G	OTHER	0.0	9999.(	-9999.(	1.010	23.4	True	NaN	NaN	VL3	VL3_0
B6-G	OTHER	0.0	9999.(	-9999.(	1.070	12.2	True	NaN	NaN	VL6	VL6_0
B8-G	OTHER	0.0	9999.(	-9999.(	1.090	17.4	True	NaN	NaN	VL8	VL8_0

```
net = pp.network.create_ieee14()
net.get_generators(attributes=['energy_source','target_p','max_p','min_p','p',
↪'voltage_level_id','bus_id'])
```

will output something like:

	energy_source	target_p	max_p	min_p	p	voltage_level_id	bus_id
id							
B1-G	OTHER	232.4	9999.0	-9999.0	NaN	VL1	VL1_0
B2-G	OTHER	40.0	9999.0	-9999.0	NaN	VL2	VL2_0
B3-G	OTHER	0.0	9999.0	-9999.0	NaN	VL3	VL3_0
B6-G	OTHER	0.0	9999.0	-9999.0	NaN	VL6	VL6_0
B8-G	OTHER	0.0	9999.0	-9999.0	NaN	VL8	VL8_0

Warning

The "generator convention" is used for the "input" columns (*target_p*, *max_p*, *min_p*, *target_v* and *target_q*) while the "load convention" is used for the ouput columns (*p* and *q*).

Most of the time, this means that *p* and *target_p* will have opposite sign. This also entails that *p* can be lower than *min_p*. Actually, the relation:
$textmin_p <= -p <=$
$textmax_p$ should hold.

pypowsybl.network.Network.get_hvdc_lines

`Network.get_hvdc_lines`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of HVDC lines.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.

- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of HVDC lines.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **converters_mode**: the mode of the converter stations. It can be either SIDE_1_RECTIFIER_SIDE_2_INVERTER or SIDE_1_INVERTER_SIDE_2_RECTIFIER
- **target_p**: active power target (in MW)
- **max_p**: the maximum of active power that can pass through the hvdc line (in MW)
- **nominal_v**: nominal voltage (in kV)
- **r**: the resistance of the hvdc line (in Ohm)
- **converter_station1_id**: at which converter station the hvdc line is connected on side "1"
- **converter_station2_id**: at which converter station the hvdc line is connected on side "2"
- **connected1**: `True` if the busbar section on side "1" is connected to a bus
- **connected2**: `True` if the busbar section on side "2" is connected to a bus
- **fictitious** (optional): `True` if the hvdc is part of the model and not of the actual network

This dataframe is indexed by the id of the hvdc lines

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_hvdc_lines()
```

will output something like:

	converters_mode	tar-get_p	max_	nom-i-nal_v	r	con-verter_statio	con-verter_statio	con-nected1	con-nected2
id									
HVD	SIDE_1_RECTIFIER_S	10.0	300.0	400.0	1.0	VSC1	VSC2	True	True
HVD	SIDE_1_RECTIFIER_S	80.0	300.0	400.0	1.0	LCC1	LCC2	True	True

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_hvdc_lines(all_attributes=True)
```

will output something like:

	converters_mode	active_power_	max_	nominal_v	r	converter_stati	converter_stati	connected	connected2
id									
HVD	SIDE_1_RECTIFIER	10.0	300.(	400.0	1.0	VSC1	VSC2	True	True
HVD	SIDE_1_RECTIFIER	80.0	300.(	400.0	1.0	LCC1	LCC2	True	True

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_hvdc_lines(attributes=['converters_mode','active_power_setpoint','nominal_v
↪','converter_station1_id','converter_station2_id','connected1','connected2'])
```

will output something like:

	converters_mode	active_power_s	nominal_v	converter_statio	converter_statio	connected1	connected2
id							
HVD(	SIDE_1_RECTIFIER_SI	10.0	400.0	VSC1	VSC2	True	True
HVD(	SIDE_1_RECTIFIER_SI	80.0	400.0	LCC1	LCC2	True	True

pypowsybl.network.Network.get_identifiables

`Network.get_identifiables`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of identifiables

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Returns

A dataframe of identifiables.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **type**: the type of the identifiable

This dataframe is indexed on the identifiable ID.

pypowsybl.network.Network.get_injections

Network.**get_injections**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of injections

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Returns
A dataframe of injections.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **type**: the type of the injection
- **voltage_level_id**: at which substation the injection is connected
- **node** (optional): node where this injection is connected, in node-breaker voltage levels
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this injection is connected
- **connected**: `True` if the injection is connected to a bus
- **bus_id**: bus where this injection is connected
- **p**: the actual active production of the injection (`NaN` if no loadflow has been computed)
- **q**: the actual reactive production of the injection (`NaN` if no loadflow has been computed)
- **i**: the current on the injection, `NaN` if no loadflow has been computed (in A)

This dataframe is indexed on the injections ID.

pypowsybl.network.Network.get_lcc_converter_stations

Network.**get_lcc_converter_stations**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of LCC converter stations.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of LCC converter stations.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **power_factor**: the power factor
- **loss_factor**: the loss factor
- **p**: active flow on the LCC converter station, `NaN` if no loadflow has been computed (in MW)
- **q**: the reactive flow on the LCC converter station, `NaN` if no loadflow has been computed (in MVAr)
- **i**: The current on the LCC converter station, `NaN` if no loadflow has been computed (in A)
- **voltage_level_id**: at which substation the LCC converter station is connected
- **bus_id**: bus where this station is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this station is connected
- **node** (optional): node where this station is connected, in node-breaker voltage levels
- **connected**: `True` if the LCC converter station is connected to a bus
- **fictitious** (optional): `True` if the LCC converter is part of the model and not of the actual network
- **hvdc_line_id**: ID of the HVDC line where the LCC converter station is connected, empty string if not connected

This dataframe is indexed by the id of the LCC converter

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_lcc_converter_stations()
```

will output something like:

.	power_factc	loss_facto	p	q	i	volt-age_level_id	bus_id	con-nected	hvdc_line_id
id									
LCC1	0.6	1.1	80.88	NaN	NaN	S1VL2	S1VL2_(	True	HVDC2
LCC2	0.6	1.1	-79.12	NaN	NaN	S3VL1	S3VL1_(	True	HVDC2

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_lcc_converter_stations(all_attributes=True)
```

will output something like:

.	power_factc	loss_facto	p	q	i	volt-age_level_id	bus_id	con-nected	hvdc_line_id
id									
LCC1	0.6	1.1	80.88	NaN	NaN	S1VL2	S1VL2_(	True	HVDC2
LCC2	0.6	1.1	-79.12	NaN	NaN	S3VL1	S3VL1_(	True	HVDC2

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_lcc_converter_stations(attributes=['p','q','i','voltage_level_id','bus_id',
↪'connected'])
```

will output something like:

.	p	q	i	voltage_level_id	bus_id	connected
id						
LCC1	80.88	NaN	NaN	S1VL2	S1VL2_0	True
LCC2	-79.12	NaN	NaN	S3VL1	S3VL1_0	True

pypowsybl.network.Network.get_lines

Network.**get_lines**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of lines data.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of lines data.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **r**: the resistance of the line (in Ohm)
- **x**: the reactance of the line (in Ohm)
- **g1**: the conductance of line at its "1" side (in Siemens)
- **b1**: the susceptance of line at its "1" side (in Siemens)
- **g2**: the conductance of line at its "2" side (in Siemens)

- **b2**: the susceptance of line at its "2" side (in Siemens)
- **p1**: the active flow on the line at its "1" side, `NaN` if no loadflow has been computed (in MW)
- **q1**: the reactive flow on the line at its "1" side, `NaN` if no loadflow has been computed (in MVAr)
- **i1**: the current on the line at its "1" side, `NaN` if no loadflow has been computed (in A)
- **p2**: the active flow on the line at its "2" side, `NaN` if no loadflow has been computed (in MW)
- **q2**: the reactive flow on the line at its "2" side, `NaN` if no loadflow has been computed (in MVAr)
- **i2**: the current on the line at its "2" side, `NaN` if no loadflow has been computed (in A)
- **voltage_level1_id**: voltage level where the line is connected, on side 1
- **voltage_level2_id**: voltage level where the line is connected, on side 2
- **bus1_id**: bus where this line is connected, on side 1
- **bus2_id**: bus where this line is connected, on side 2
- **bus_breaker_bus1_id** (optional): bus of the bus-breaker view where this line is connected, on side 1
- **bus_breaker_bus2_id** (optional): bus of the bus-breaker view where this line is connected, on side 2
- **node1** (optional): node where this line is connected on side 1, in node-breaker voltage levels
- **node2** (optional): node where this line is connected on side 2, in node-breaker voltage levels
- **connected1**: `True` if the side "1" of the line is connected to a bus
- **connected2**: `True` if the side "2" of the line is connected to a bus
- **fictitious** (optional): `True` if the line is part of the model and not of the actual network
- **selected_limits_group_1** (optional): Name of the selected operational limits group selected for side 1
- **selected_limits_group_2** (optional): Name of the selected operational limits group selected for side 2

This dataframe is indexed by the id of the lines.

Examples

```
net = pp.network.create_ieee14()
net.get_lines()
```

will output something like:

	r	x	g1	b1	g2	b2	p1	q1	i1	p2	q2	i2	volt-age_	volt-age_	bus	bus	con-nec	con-nected2
id																		
L1-2-1	0.00	0.00	0.0	2.64	0.0	2.64	NaN	NaN	NaN	NaN	NaN	NaN	VL1	VL2	VL1	VL2	True	True
L1-5-1	0.00	0.00	0.0	2.46	0.0	2.46	NaN	NaN	NaN	NaN	NaN	NaN	VL1	VL5	VL1	VL5	True	True

```
net = pp.network.create_ieee14()
net.get_lines(all_attributes=True)
```

will output something like:

	r	x	g1	b1	g2	b2	p1	q1	i1	p2	q2	i2	volt-age	volt-age	bus	bus	con-nec	con-nected2
id																		
L1-2-1	0.00	0.00	0.0	2.64	0.0	2.64	NaN	NaN	NaN	NaN	NaN	NaN	VL1	VL2	VL1	VL2	True	True
L1-5-1	0.00	0.00	0.0	2.46	0.0	2.46	NaN	NaN	NaN	NaN	NaN	NaN	VL1	VL5	VL1	VL5	True	True

```
net = pp.network.create_ieee14()
net.get_lines(attributes=['p1','q1','i1','p2','q2','i2','voltage_level1_id',
↪'voltage_level2_id','bus1_id','bus2_id','connected1','connected2'])
```

will output something like:

	p1	q1	i1	p2	q2	i2	volt-age_level1	volt-age_level2	bus1_	bus2_	con-nected1	con-nected2
id												
L1-2-1	NaN	NaN	NaN	NaN	NaN	NaN	VL1	VL2	VL1_(	VL2_(	True	True
L1-5-1	NaN	NaN	NaN	NaN	NaN	NaN	VL1	VL5	VL1_(	VL5_(	True	True

pypowsybl.network.Network.get_loads

`Network.get_loads`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of loads.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

the loads dataframe

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **type**: type of load
- **p0**: the active load consumption setpoint (MW)
- **q0**: the reactive load consumption setpoint (MVAr)
- **p**: the result active load consumption, it is `NaN` is not loadflow has been computed (MW)
- **q**: the result reactive load consumption, it is `NaN` is not loadflow has been computed (MVAr)
- **i**: the current on the load, `NaN` if no loadflow has been computed (in A)
- **voltage_level_id**: at which substation this load is connected
- **bus_id**: bus where this load is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this load is connected
- **node** (optional): node where this load is connected, in node-breaker voltage levels
- **connected**: `True` if the load is connected to a bus
- **fictitious** (optional): `True` if the load is part of the model and not of the actual network

This dataframe is indexed on the load ID.

Examples

```
net = pp.network.create_ieee14()
net.get_loads()
```

will output something like:

	type	p0	q0	p	q	voltage_level_id	bus_id	connected
id								
B2-L	UNDEFINED	21.7	12.7	NaN	NaN	VL2	VL2_0	True
B3-L	UNDEFINED	94.2	19.0	NaN	NaN	VL3	VL3_0	True
B4-L	UNDEFINED	47.8	-3.9	NaN	NaN	VL4	VL4_0	True
B5-L	UNDEFINED	7.6	1.6	NaN	NaN	VL5	VL5_0	True
B6-L	UNDEFINED	11.2	7.5	NaN	NaN	VL6	VL6_0	True
B9-L	UNDEFINED	29.5	16.6	NaN	NaN	VL9	VL9_0	True
B10-L	UNDEFINED	9.0	5.8	NaN	NaN	VL10	VL10_0	True
B11-L	UNDEFINED	3.5	1.8	NaN	NaN	VL11	VL11_0	True
B12-L	UNDEFINED	6.1	1.6	NaN	NaN	VL12	VL12_0	True
B13-L	UNDEFINED	13.5	5.8	NaN	NaN	VL13	VL13_0	True
B14-L	UNDEFINED	14.9	5.0	NaN	NaN	VL14	VL14_0	True

```
net = pp.network.create_ieee14()
net.get_loads(all_attributes=True)
```

will output something like:

	type	p0	q0	p	q	voltage_level_id	bus_id	connected
id								
B2-L	UNDEFINED	21.7	12.7	NaN	NaN	VL2	VL2_0	True
B3-L	UNDEFINED	94.2	19.0	NaN	NaN	VL3	VL3_0	True
B4-L	UNDEFINED	47.8	-3.9	NaN	NaN	VL4	VL4_0	True
B5-L	UNDEFINED	7.6	1.6	NaN	NaN	VL5	VL5_0	True
B6-L	UNDEFINED	11.2	7.5	NaN	NaN	VL6	VL6_0	True
B9-L	UNDEFINED	29.5	16.6	NaN	NaN	VL9	VL9_0	True
B10-L	UNDEFINED	9.0	5.8	NaN	NaN	VL10	VL10_0	True
B11-L	UNDEFINED	3.5	1.8	NaN	NaN	VL11	VL11_0	True
B12-L	UNDEFINED	6.1	1.6	NaN	NaN	VL12	VL12_0	True
B13-L	UNDEFINED	13.5	5.8	NaN	NaN	VL13	VL13_0	True
B14-L	UNDEFINED	14.9	5.0	NaN	NaN	VL14	VL14_0	True

```
net = pp.network.create_ieee14()
net.get_loads(attributes=['type','p','q','voltage_level_id','bus_id','connected'])
```

will output something like:

	type	p	q	voltage_level_id	bus_id	connected
id						
B2-L	UNDEFINED	NaN	NaN	VL2	VL2_0	True
B3-L	UNDEFINED	NaN	NaN	VL3	VL3_0	True
B4-L	UNDEFINED	NaN	NaN	VL4	VL4_0	True
B5-L	UNDEFINED	NaN	NaN	VL5	VL5_0	True
B6-L	UNDEFINED	NaN	NaN	VL6	VL6_0	True
B9-L	UNDEFINED	NaN	NaN	VL9	VL9_0	True
B10-L	UNDEFINED	NaN	NaN	VL10	VL10_0	True
B11-L	UNDEFINED	NaN	NaN	VL11	VL11_0	True
B12-L	UNDEFINED	NaN	NaN	VL12	VL12_0	True
B13-L	UNDEFINED	NaN	NaN	VL13	VL13_0	True
B14-L	UNDEFINED	NaN	NaN	VL14	VL14_0	True

pypowsybl.network.Network.get_linear_shunt_compensator_sections

Network.**get_linear_shunt_compensator_sections**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of shunt compensators sections for linear model.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **g_per_section**: the conductance per section in S
- **b_per_section**: the susceptance per section in S
- **max_section_count**: the maximum number of sections

This dataframe is indexed by the shunt compensator ID.

Returns
A dataframe of linear models of shunt compensators.

Parameters

- **all_attributes** (*bool*)
- **attributes** (*List[str] | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
DataFrame

pypowsybl.network.Network.get_non_linear_shunt_compensator_sections

Network.**get_non_linear_shunt_compensator_sections**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of shunt compensators sections for non linear model.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Return type
DataFrame

Notes

The resulting dataframe will have the following columns:

- **g**: the accumulated conductance in S if the section and all the previous ones are activated.
- **b**: the accumulated susceptance in S if the section and all the previous ones are activated

This dataframe is multi-indexed, by the tuple (id of shunt, section number).

Returns
A dataframe of non linear model shunt compensators sections.

Parameters

- **all_attributes** (*bool*)

- **attributes** (*List[str] | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
DataFrame

pypowsybl.network.Network.get_operational_limits

Network.**get_operational_limits**(*all_attributes=False*, *attributes=None*, *show_inactive_sets=False*)

Get the list of operational limits.

The resulting dataframe, depending on the parameters, will have some of the following columns:

- **element_id**: Identifier of the network element on which this limit applies (could be for example a line or a transformer)
- **element_type**: Type of the network element on which this limit applies (LINE, TWO_WINDINGS_TRANSFORMER, THREE_WINDINGS_TRANSFORMER, BOUNDARY_LINE)
- **side**: The side of the element on which this limit applies (ONE, TWO, THREE)
- **name**: The name of the limit
- **type**: The type of the limit (CURRENT, ACTIVE_POWER, APPARENT_POWER)
- **value**: The value of the limit
- **acceptable_duration**: The duration, in seconds, for which the element can securely be operated under the limit value. By convention, the value -1 represents an infinite duration.
- **fictitious** (optional): *True* if this limit is fictitious
- **group_name** (optional): The name of the operational limit group this limit is in
- **selected** (optional): *True* if this limit's operational group is the selected one

The index of the dataframe is composed of the columns **element_id**, **side**, **type**, **acceptable_duration** and **group_name**.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **show_inactive_sets** (*bool*) – flag to choose whether inactive limit sets should also be included in the dataframe

Returns
All limits on the network

Return type
DataFrame

pypowsybl.network.Network.get_phase_tap_changer_steps

Network.**get_phase_tap_changer_steps**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of phase tap changer steps.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of phase tap changer steps.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **side**: the phase tap changer side in case of a belonging to a 3 windings transformer, empty for a 2 windings transformer
- **rho**: The voltage ratio in per unit of the rated voltages (in per unit)
- **alpha**: the angle difference (in degree)
- **r**: The resistance deviation in percent of nominal value (%)
- **x**: The reactance deviation in percent of nominal value (%)
- **g**: The conductance deviation in percent of nominal value (%)
- **b**: The susceptance deviation in percent of nominal value (%)

This dataframe is index by the id of the transformer and the position of the phase tap changer step

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_phase_tap_changer_steps()
```

will output something like:

		side	rho	alpha	r	x	g	b
id	position							
TWT	0		1.0	-42.80	39.784730	29.784725	0.0	0.0
	1		1.0	-40.18	31.720245	21.720242	0.0	0.0
	2		1.0	-37.54	23.655737	13.655735	0.0	0.0
…	…	…	…	…	…	…	…	…

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_phase_tap_changer_steps(all_attributes=True)
```

will output something like:

		side	rho	alpha	r	x	g	b
id	position							
TWT	0		1.0	-42.80	39.784730	29.784725	0.0	0.0
	1		1.0	-40.18	31.720245	21.720242	0.0	0.0
	2		1.0	-37.54	23.655737	13.655735	0.0	0.0
…	…	…	…	…	…	…	…	…

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_phase_tap_changer_steps(attributes=['rho','r','x'])
```

will output something like:

		side	rho	r	x
id	position				
TWT	0		1.0	39.784730	29.784725
	1		1.0	31.720245	21.720242
	2		1.0	23.655737	13.655735
…	…	…	…	…	…

pypowsybl.network.Network.get_phase_tap_changers

Network.**get_phase_tap_changers**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Create a phase tap changers:class:*~pandas.DataFrame*.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

the phase tap changers dataframe

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **side**: the phase tap changer side in case of a belonging to a 3 windings transformer, empty for a 2 windings transformer

- **tap**: the current tap position (input of a loadflow)
- **solved_tap_position**: the tap position obtained after running a loadflow (NaN before any computation)
- **low_tap**: the low tap position (usually 0, but could be different depending on the data origin)
- **high_tap**: the high tap position
- **step_count**: the count of taps, should be equal to (high_tap - low_tap)
- **oltc**: true if the tap changer has on-load regulation capability
- **regulating**: true if the phase shifter is in regulation
- **regulation_mode**: regulation mode, among CURRENT_LIMITER and ACTIVE_POWER_CONTROL
- **regulation_value**: the target value, in A or MW, depending on regulation_mode
- **target_deadband**: the regulation deadband around the target value
- **regulating_bus_id**: the bus where the phase shifter regulates
- **regulated_side** (optional): the side bus where the phase shifter regulates current or active power

This dataframe is indexed by the id of the transformer

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_phase_tap_changers()
```

will output something like:

	side	tap	low_	solved_ta	high_	step_	oltc	reg-u-lat-ing	regula-tion_mod	reg-ula-tion_val	tar-get_dea	reg-ulat-ing_bus_id
id												
TWT		15	0	NaN	32	33	True	False	CUR-RENT_LI	NaN	NaN	S1VL1_0

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_phase_tap_changers(all_attributes=True)
```

will output something like:

	side	tap	low_	solved_	high_	step_	oltc	reg-u-lat-ing	reg-ula-tion_mo	reg-ula-tion_va	tar-get_de	reg-ulat-ing_bu	reg-u-lated_side
id													
TWT		15	0	NaN	32	33	True	False	CUR-RENT_L	NaN	NaN	S1VL1_	ONE

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_phase_tap_changers(attributes=['tap','low_tap','high_tap','step_count',
↪'regulating_bus_id'])
```

will output something like:

	side	tap	low_tap	high_tap	step_count	regulating_bus_id
id						
TWT		15	0	32	33	S1VL1_0

pypowsybl.network.Network.get_ratio_tap_changer_steps

Network.**get_ratio_tap_changer_steps**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of ratio tap changer steps.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of ratio tap changer steps.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **side**: the ratio tap changer side in case of a belonging to a 3 windings transformer, empty for a 2 windings transformer
- **rho**: The voltage ratio in per unit of the rated voltages (in per unit)
- **r**: The resistance deviation in percent of nominal value (%)
- **x**: The reactance deviation in percent of nominal value (%)
- **g**: The conductance deviation in percent of nominal value (%)
- **b**: The susceptance deviation in percent of nominal value (%)

This dataframe is index by the id of the transformer and the position of the ratio tap changer step

Examples

```
net = pp.network.create_eurostag_tutorial_example1_network()
net.get_ratio_tap_changer_steps()
```

will output something like:

		side	rho	r	x	g	b
id	position						
NHV2_NLOAD	0		0.850567	0.0	0.0	0.0	0.0
	1		1.000667	0.0	0.0	0.0	0.0
	2		1.150767	0.0	0.0	0.0	0.0

```
net = pp.network.create_eurostag_tutorial_example1_network()
net.get_ratio_tap_changer_steps(all_attributes=True)
```

will output something like:

		side	rho	r	x	g	b
id	position						
NHV2_NLOAD	0		0.850567	0.0	0.0	0.0	0.0
	1		1.000667	0.0	0.0	0.0	0.0
	2		1.150767	0.0	0.0	0.0	0.0

```
net = pp.network.create_eurostag_tutorial_example1_network()
net.get_ratio_tap_changer_steps(attributes=['rho','r','x'])
```

will output something like:

		side	rho	r	x
id	position				
NHV2_NLOAD	0		0.850567	0.0	0.0
	1		1.000667	0.0	0.0
	2		1.150767	0.0	0.0

pypowsybl.network.Network.get_ratio_tap_changers

Network.**get_ratio_tap_changers**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Create a ratio tap changers:class:*~pandas.DataFrame*.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

the ratio tap changers dataframe

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **side**: the ratio tap changer side in case of a belonging to a 3 windings transformer, empty for a 2 windings transformer
- **tap**: the current tap position (input of a loadflow)
- **solved_tap_position**: the tap position obtained after running a loadflow (`NaN` before any computation)
- **low_tap**: the low tap position (usually 0, but could be different depending on the data origin)
- **high_tap**: the high tap position
- **step_count**: the count of taps, should be equal to (high_tap - low_tap)
- **oltc**: true if the tap changer has on-load regulation capability
- **regulating**: true if the tap changer is in regulation
- **target_v**: the target voltage in kV, if the tap changer is in regulation
- **target_deadband**: the regulation deadband around the target voltage, in kV
- **regulating_bus_id**: the bus where the tap changer regulates voltage
- **regulated_side** (optional): the side where the tap changer regulates voltage (redundant with regulating_bus_id)

This dataframe is indexed by the id of the transformer

Examples

```
net = pp.network.create_eurostag_tutorial_example1_network()
net.get_ratio_tap_changers()
```

will output something like:

	side	tap	solved_tap_	low_t	high_	step_c	oltc	reg-ulat-ing	tar-get_v	tar-get_deadb	regulat-ing_bus_id
id											
NHV2_N		1	NaN	0	2	3	True	True	158.0	0.0	VL-LOAD_0

```
net = pp.network.create_eurostag_tutorial_example1_network()
net.get_ratio_tap_changers(all_attributes=True)
```

will output something like:

	side	tap	solved_ta	low_	high_	step_(	oltc	reg-u-lat-ing	tar-get_\	tar-get_dea(	reg-ulat-ing_bus_	regu-lated_side
id												
NHV2_1		1	nan	0	2	3	True	True	158.0	0.0	VL-LOAD_0	TWO

```
net = pp.network.create_eurostag_tutorial_example1_network()
net.get_ratio_tap_changers(attributes=['tap','low_tap','high_tap','step_count',
↪'target_v','regulating_bus_id'])
```

will output something like:

	side	tap	low_tap	high_tap	step_count	target_v	regulating_bus_id
id							
NHV2_NLOAD		1	0	2	3	158.0	VLLOAD_0

pypowsybl.network.Network.get_reactive_capability_curve_points

Network.**get_reactive_capability_curve_points**(*all_attributes=False*, *attributes=None*)

Get a dataframe of reactive capability curve points.

For each generator, the min/max reactive capabilities can be represented as curves. This dataframe describes those curves as a list of points, which associate a min and a max value of Q to a given value of P.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.

Returns

A dataframe of reactive capability curve points.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **num**: the point position in the curve description (starts 0 for a given generator)
- **p**: the active power of the point, in MW
- **min_q**: the minimum value of reactive power, in MVar, for this value of P
- **max_q**: the maximum value of reactive power, in MVar, for this value of P

This dataframe is indexed on the generator ID.

pypowsybl.network.Network.get_shunt_compensators

Network.**get_shunt_compensators**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of shunt compensators.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of shunt compensators.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **model_type**:
- **max_section_count**: The maximum number of sections that may be switched on
- **section_count**: The number of section in service (input for the computation)
- **solved_section_count**: The number of section in service (output of a computation, NaN before any loadflow)
- **p**: the active flow on the shunt, NaN if no loadflow has been computed (in MW)
- **q**: the reactive flow on the shunt, NaN if no loadflow has been computed (in MVAr)
- **i**: the current in the shunt, NaN if no loadflow has been computed (in A)
- **voltage_level_id**: at which substation the shunt is connected
- **bus_id**: bus where this shunt is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this shunt is connected
- **node** (optional): node where this shunt is connected, in node-breaker voltage levels
- **connected**: True if the shunt is connected to a bus
- **fictitious** (optional): True if the shunt is part of the model and not of the actual network

This dataframe is indexed by the id of the shunt compensators

Examples

```
net = pp.network.create_ieee14()
net.get_shunt_compensators()
```

will output something like:

	model_typ	max_section_cc	sec-tion_count	p	q	i	volt-age_level_id	bus_ic	con-nected
id									
B9-SH	LINEAR	1	1	NaN	NaN	NaN	VL9	VL9_0	True

```
net = pp.network.create_ieee14()
net.get_shunt_compensators(all_attributes=True)
```

will output something like:

	model_typ	max_section_cc	sec-tion_count	p	q	i	volt-age_level_id	bus_ic	con-nected
id									
B9-SH	LINEAR	1	1	NaN	NaN	NaN	VL9	VL9_0	True

```
net = pp.network.create_ieee14()
net.get_shunt_compensators(attributes=['model_type','p','q','i','voltage_level_id',
↪'bus_id','connected'])
```

will output something like:

	model_type	p	q	i	voltage_level_id	bus_id	connected
id							
B9-SH	LINEAR	NaN	NaN	NaN	VL9	VL9_0	True

pypowsybl.network.Network.get_static_var_compensators

Network.**get_static_var_compensators**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of static var compensators.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of static var compensators.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **b_min**: the minimum susceptance
- **b_max**: the maximum susceptance
- **target_v**: The voltage setpoint
- **target_q**: The reactive power setpoint
- **regulation_mode**: The regulation mode
- **regulated_element_id**: The ID of the network element where voltage is regulated
- **regulated_bus_id**: the ID of the bus from bus view where voltage is regulated
- **regulated_bus_breaker_bus_id**: the ID of the bus from bus/breaker view where voltage is regulated
- **p**: active flow on the var compensator, `NaN` if no loadflow has been computed (in MW)
- **q**: the reactive flow on the var compensator, `NaN` if no loadflow has been computed (in MVAr)
- **i**: The current on the var compensator, `NaN` if no loadflow has been computed (in A)
- **voltage_level_id**: at which substation the var compensator is connected
- **bus_id**: bus where this SVC is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this SVC is connected
- **node** (optional): node where this SVC is connected, in node-breaker voltage levels
- **connected**: `True` if the var compensator is connected to a bus
- **fictitious** (optional): `True` if the var compensator is part of the model and not of the actual network

This dataframe is indexed by the id of the var compensator

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_static_var_compensators()
```

will output something like:

	b_mi	b_ma	tar-get_v	tar-get_q	regula-tion_mod	regu-lated_eleme	p	q	i	volt-age_leve	bus_i	con-nected
id												
SVC	-0.05	0.05	400.0	NaN	VOLT-AGE	SVC	NaN	-12.54	NaN	S4VL1	S4VL	True

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_static_var_compensators(all_attributes=True)
```

will output something like:

	b_mi	b_ma	volt-age_set	reac-tive_power	reg-ula-tion_mo	regu-lated_eler	p	q	i	volt-age_lev	bus_	con-nected
id												
SVC	-0.05	0.05	400.0	NaN	VOLT-AGE	SVC	NaN	-12.54	NaN	S4VL1	S4VI	True

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_static_var_compensators(attributes=['p','q','i','voltage_level_id','bus_id',
↪'connected'])
```

will output something like:

	p	q	i	voltage_level_id	bus_id	connected
id						
SVC	NaN	-12.5415	NaN	S4VL1	S4VL1_0	True

pypowsybl.network.Network.get_substations

`Network.get_substations`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get substations `DataFrame`.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of substations.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **name**: the name of the substations
- **TSO**: the TSO which the substation belongs to
- **geo_tags**: additional geographical information about the substation
- **country**: the country which the substation belongs to
- **fictitious** (optional): `True` if the substation is part of the model and not of the actual network

This dataframe is indexed on the substation ID.

pypowsybl.network.Network.get_switches

Network.**get_switches**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of switches.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns

A dataframe of switches.

Return type

DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **kind**: the kind of switch
- **open**: the open status of the switch
- **retained**: the retain status of the switch
- **voltage_level_id**: at which substation the switch is connected
- **bus_breaker_bus1_id** (optional): bus where this switch is connected on side 1, in bus-breaker voltage levels
- **bus_breaker_bus1_id** (optional): bus where this switch is connected on side 1, in bus-breaker voltage levels
- **node1** (optional): node where this switch is connected on side 1, in node-breaker voltage levels
- **node2** (optional): node where this switch is connected on side 2, in node-breaker voltage levels
- **fictitious** (optional): `True` if the switch is part of the model and not of the actual network

This dataframe is indexed by the id of the switches

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_switches()
```

will output something like:

	kind	open	retained	voltage_level_id
id				
S1VL1_BBS_LD1_DISCONNECTOR	DISCONNECTOR	False	False	S1VL1
S1VL1_LD1_BREAKER	BREAKER	False	True	S1VL1
S1VL1_BBS_TWT_DISCONNECTOR	DISCONNECTOR	False	False	S1VL1
S1VL1_TWT_BREAKER	BREAKER	False	True	S1VL1
S1VL2_BBS1_TWT_DISCONNECTOR	DISCONNECTOR	False	False	S1VL2
S1VL2_BBS2_TWT_DISCONNECTOR	DISCONNECTOR	True	False	S1VL2
S1VL2_TWT_BREAKER	BREAKER	False	True	S1VL2
S1VL2_BBS1_VSC1_DISCONNECTOR	DISCONNECTOR	True	False	S1VL2
…	…	…	…	…

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_switches(all_attributes=True)
```

will output something like:

	kind	open	retained	voltage_level_id
id				
S1VL1_BBS_LD1_DISCONNECTOR	DISCONNECTOR	False	False	S1VL1
S1VL1_LD1_BREAKER	BREAKER	False	True	S1VL1
S1VL1_BBS_TWT_DISCONNECTOR	DISCONNECTOR	False	False	S1VL1
S1VL1_TWT_BREAKER	BREAKER	False	True	S1VL1
S1VL2_BBS1_TWT_DISCONNECTOR	DISCONNECTOR	False	False	S1VL2
S1VL2_BBS2_TWT_DISCONNECTOR	DISCONNECTOR	True	False	S1VL2
S1VL2_TWT_BREAKER	BREAKER	False	True	S1VL2
S1VL2_BBS1_VSC1_DISCONNECTOR	DISCONNECTOR	True	False	S1VL2
…	…	…	…	…

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_switches(attributes=['kind','open','nominal_v','voltage_level_id'])
```

will output something like:

	kind	open	voltage_level_id
id			
S1VL1_BBS_LD1_DISCONNECTOR	DISCONNECTOR	False	S1VL1
S1VL1_LD1_BREAKER	BREAKER	False	S1VL1
S1VL1_BBS_TWT_DISCONNECTOR	DISCONNECTOR	False	S1VL1
S1VL1_TWT_BREAKER	BREAKER	False	S1VL1
S1VL2_BBS1_TWT_DISCONNECTOR	DISCONNECTOR	False	S1VL2
S1VL2_BBS2_TWT_DISCONNECTOR	DISCONNECTOR	True	S1VL2
S1VL2_TWT_BREAKER	BREAKER	False	S1VL2
S1VL2_BBS1_VSC1_DISCONNECTOR	DISCONNECTOR	True	S1VL2
…	…	…	…

pypowsybl.network.Network.get_terminals

Network.**get_terminals**(*all_attributes=False*, *attributes=None*)

Get a dataframe of terminal

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.

Returns
A dataframe of terminals.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **voltage_level_id**: voltage level where the terminal is connected
- **bus_id**: bus where this terminal is
- **element_side**: if it is a terminal of a branch it will indicate his side else it is “”

This dataframe is indexed on the element ID of the terminal.

pypowsybl.network.Network.get_voltage_levels

Network.**get_voltage_levels**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of voltage levels.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of voltage levels.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **substation_id**: at which substation the voltage level belongs
- **nominal_v**: The nominal voltage

- **high_voltage_limit**: the high voltage limit
- **low_voltage_limit**: the low voltage limit
- **fictitious** (optional): `True` if the voltage level is part of the model and not of the actual network
- **topology_kind** (optional): the voltage level topology kind (NODE_BREAKER or BUS_BREAKER)

This dataframe is indexed by the id of the voltage levels

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_voltage_levels()
```

will output something like:

	substation_id	nominal_v	high_voltage_limit	low_voltage_limit
id				
S1VL1	S1	225.0	240.0	220.0
S1VL2	S1	400.0	440.0	390.0
S2VL1	S2	400.0	440.0	390.0
S3VL1	S3	400.0	440.0	390.0
S4VL1	S4	400.0	440.0	390.0

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_voltage_levels(all_attributes=True)
```

will output something like:

	substation_id	nominal_v	high_voltage_limit	low_voltage_limit
id				
S1VL1	S1	225.0	240.0	220.0
S1VL2	S1	400.0	440.0	390.0
S2VL1	S2	400.0	440.0	390.0
S3VL1	S3	400.0	440.0	390.0
S4VL1	S4	400.0	440.0	390.0

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_voltage_levels(attributes=['substation_id','nominal_v'])
```

will output something like:

	substation_id	nominal_v
id		
S1VL1	S1	225.0
S1VL2	S1	400.0
S2VL1	S2	400.0
S3VL1	S3	400.0
S4VL1	S4	400.0

pypowsybl.network.Network.get_vsc_converter_stations

Network.**get_vsc_converter_stations**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of VSC converter stations.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of VCS converter stations.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **loss_factor**: correspond to the loss of power due to ac dc conversion
- **target_v**: The voltage setpoint
- **target_q**: The reactive power setpoint
- **max_q**: the maximum reactive value for the VSC converter station only if reactive_limits_kind is MIN_MAX (MVar)
- **min_q**: the minimum reactive value for the VSC converter station only if reactive_limits_kind is MIN_MAX (MVar)
- **max_q_at_target_p** (optional): the maximum reactive value for the VSC converter station for the target p specified (MVar)
- **min_q_at_target_p** (optional): the minimum reactive value for the VSC converter station for the target p specified (MVar)
- **max_q_at_p** (optional): the maximum reactive value for the VSC converter station at current p (MVar)
- **min_q_at_p** (optional): the minimum reactive value for the VSC converter station at current p (MVar)
- **reactive_limits_kind**: type of the reactive limit of the VSC converter station (can be MIN_MAX, CURVE or NONE)
- **voltage_regulator_on**: The voltage regulator status
- **regulated_element_id**: The ID of the network element where voltage is regulated
- **regulated_bus_id**: the ID of the bus from bus view where voltage is regulated
- **regulated_bus_breaker_bus_id**: the ID of the bus from bus/breaker view where voltage is regulated
- **p**: active flow on the VSC converter station, NaN if no loadflow has been computed (in MW)
- **q**: the reactive flow on the VSC converter station, NaN if no loadflow has been computed (in MVAr)
- **i**: The current on the VSC converter station, NaN if no loadflow has been computed (in A)

- **voltage_level_id**: at which substation the VSC converter station is connected
- **bus_id**: bus where this station is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this station is connected
- **node** (optional): node where this station is connected, in node-breaker voltage levels
- **connected**: `True` if the VSC converter station is connected to a bus
- **fictitious** (optional): `True` if the VSC converter is part of the model and not of the actual network
- **hvdc_line_id**: ID of the HVDC line where the VSC converter station is connected, empty string if not connected

This dataframe is indexed by the id of the VSC converter

Examples

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_vsc_converter_stations()
```

will output something like:

	loss_	volt-age_se	reac-tive_powe	volt-age_regu	regu-lated_ele	p	q	i	volt-age_le	bus_	con-necte	hvdc_line_id
id												
VSC	1.1	400.0	500.0	True	VSC1	10.11	-512.(	739.2	S1VL2	S1VI	True	HVDC1
VSC	1.1	0.0	120.0	False	VSC2	-9.89	-120.(	170.0	S2VL1	S2VI	True	HVDC1

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_vsc_converter_stations(all_attributes=True)
```

will output something like:

	loss_f	tar-get_	tar-get_(	volt-age_regul	regu-lated_elem	p	q	i	volt-age_lev	bus_	con-necte	hvdc_line_id
id												
VSC	1.1	400.(	500.(	True	VSC1	10.11	-512.0	739.26	S1VL2	S1VL	True	HVDC1
VSC	1.1	0.0	120.(	False	VSC2	-9.89	-120.0	170.03	S2VL1	S2VL	True	HVDC1

```
net = pp.network.create_four_substations_node_breaker_network()
net.get_vsc_converter_stations(attributes=['p','q','i','voltage_level_id','bus_id',
↪'connected'])
```

will output something like:

	p	q	i	voltage_level_id	bus_id	connected
id						
VSC1	10.11	-512.0814	739.269871	S1VL2	S1VL2_0	True
VSC2	-9.89	-120.0000	170.031658	S2VL1	S2VL1_0	True

pypowsybl.network.Network.get_tie_lines

`Network.get_tie_lines`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of tie lines.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
A dataframe of tie lines.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **boundary_line1_id**: The ID of the first boundary line
- **boundary_line2_id**: The ID of the second boundary line
- **dangling_line1_id**: deprecated for **boundary_line1_id**
- **dangling_line2_id**: deprecated for **boundary_line2_id**
- **pairing_key**: the pairing key of the tie line, obtained from the boundary lines.
- **ucte_xnode_code**: deprecated for **pairing_key**.
- **connected1**: `True` if the dangling line 1 is connected to a bus
- **connected2**: `True` if the dangling line 2 is connected to a bus
- **fictitious** (optional): `True` if the tie line is part of the model and not of the actual network

This dataframe is indexed by the id of the tie lines

pypowsybl.network.Network.get_grounds

`Network.get_grounds`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of grounds.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false

- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
the grounds dataframe

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **voltage_level_id**: at which substation this ground is connected
- **bus_id**: bus where this ground is connected
- **bus_breaker_bus_id** (optional): bus of the bus-breaker view where this ground is connected
- **node** (optional): node where this ground is connected, in node-breaker voltage levels
- **connected**: `True` if the ground is connected to a bus

This dataframe is indexed on the ground ID.

pypowsybl.network.Network.get_sub_networks

`Network.get_sub_networks`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of sub networks

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Returns
A dataframe of sub networks.

Return type
DataFrame

pypowsybl.network.Network.get_sub_network

`Network.get_sub_network`(*sub_network_id*)

Get a sub network from its parent network.

Parameters
sub_network_id (*str*) – the id of the sub network

Returns
The sub network.

Return type
Network

pypowsybl.network.Network.detach

`Network.detach()`

Detach a sub network from its parent network.

Return type
None

pypowsybl.network.Network.get_dc_nodes

`Network.get_dc_nodes(all_attributes=False, attributes=None, **kwargs)`

Get a dataframe of dc nodes.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
the dc nodes dataframe

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **nominal_v**: dc node nominal voltage
- **dc_bus_id**: at which dc bus the dc node belongs
- **fictitious** (optional): `True` if the area is part of the model and not of the actual network

This dataframe is indexed on the dc node ID.

pypowsybl.network.Network.get_dc_lines

`Network.get_dc_lines(all_attributes=False, attributes=None, **kwargs)`

Get a dataframe of DC lines

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false

- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Returns
A dataframe of DC lines.

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **dc_node1_id**: dc node where this dc line is connected, on side 1
- **dc_node2_id**: dc node where this dc line is connected, on side 2
- **p1**: the active flow on the dc line at its "1" side, `NaN` if no loadflow has been computed (in MW)
- **i1**: the current on the dc line at its "1" side, `NaN` if no loadflow has been computed (in A)
- **p2**: the active flow on the dc line at its "2" side, `NaN` if no loadflow has been computed (in MW)
- **i2**: the current on the dc line at its "2" side, `NaN` if no loadflow has been computed (in A)
- **fictitious** (optional): `True` if the area is part of the model and not of the actual network

This dataframe is indexed on the dc line ID.

pypowsybl.network.Network.get_voltage_source_converters

`Network.get_voltage_source_converters`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of voltage source converters.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
the voltage source converters dataframe

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **voltage_level_id**: at which substation the converter is connected
- **bus1_id**: bus where this converter is connected, on side 1
- **bus2_id** (optional): bus where this converter is connected, on side 2
- **dc_node1_id**: dc node where this converter is connected, on side 1
- **dc_node2_id**: dc node where this converter is connected, on side 2
- **regulated_element_id** (optional): which element of the network is regulating PCC, needed if bus2_id is set
- **dc_connected1**: `True` if the converter is connected to a dc node, side 1
- **dc_connected2**: `True` if the converter is connected to a dc node, side 2
- **voltage_regulator_on**: the voltage regulator status
- **control_mode**: the control mode of the converter
- **target_p**: the active power setpoint
- **target_q**: the reactive power setpoint
- **target_v_dc**: the DC voltage setpoint
- **target_v_ac**: the AC voltage setpoint
- **idle_loss**: the idle loss coefficient
- **switching_loss**: the switching loss coefficient
- **resistive_loss**: the resistive loss coefficient
- **p_ac**: the AC active flow on the converter, `NaN` if no loadflow has been computed (in MW)
- **q_ac**: the AC reactive flow on the converter, `NaN` if no loadflow has been computed (in MVAr)
- **p_dc1**: the DC flow on the converter, side 1 `NaN` if no loadflow has been computed (in MW)
- **p_dc2**: the DC flow on the converter, side 2 `NaN` if no loadflow has been computed (in MW)
- **fictitious** (optional): `True` if the area is part of the model and not of the actual network

This dataframe is indexed on the converter ID.

pypowsybl.network.Network.get_dc_grounds

`Network.get_dc_grounds`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of dc grounds.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
the dc grounds dataframe

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **dc_node_id**: dc node identifier
- **r**: dc ground resistance

This dataframe is indexed on the dc ground ID.

pypowsybl.network.Network.get_dc_buses

`Network.get_dc_buses`(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of dc buses.

Parameters

- **all_attributes** (*bool*) – flag for including all attributes in the dataframe, default is false
- **attributes** (*List[str] | None*) – attributes to include in the dataframe. The 2 parameters are mutually exclusive. If no parameter is specified, the dataframe will include the default attributes.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Returns
the dc buses dataframe

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **connected_component**: The connected component to which the dc bus belongs
- **dc_component**: The dc component to which the dc bus belongs
- **v**: dc bus voltage
- **fictitious** (optional): `True` if the area is part of the model and not of the actual network

This dataframe is indexed on the dc bus ID.

Bus/Breaker or Node/Breaker topology description of a given voltage level can be retrieved using the following getters:

Network.get_bus_breaker_topology	Get the bus breaker description of the topology of a voltage level.
Network.get_node_breaker_topology	Get the node breaker description of the topology of a voltage level.

pypowsybl.network.Network.get_bus_breaker_topology

Network.**get_bus_breaker_topology**(*voltage_level_id*)

Get the bus breaker description of the topology of a voltage level.

Parameters
voltage_level_id (*str*) – id of the voltage level

Returns
The bus breaker description of the topology of the voltage level

Return type
BusBreakerTopology

pypowsybl.network.Network.get_node_breaker_topology

Network.**get_node_breaker_topology**(*voltage_level_id*)

Get the node breaker description of the topology of a voltage level.

Parameters
voltage_level_id (*str*) – id of the voltage level

Returns
The node breaker description of the topology of the voltage level

Return type
NodeBreakerTopology

These getters return an object of the following classes:

BusBreakerTopology	Bus-breaker representation of the topology of a voltage level.
NodeBreakerTopology	Node-breaker representation of the topology of a voltage level.

pypowsybl.network.BusBreakerTopology

class BusBreakerTopology(*network_handle*, *voltage_level_id*)

Bus-breaker representation of the topology of a voltage level.

The topology is actually represented as a graph, where vertices are buses while edges are switches (breakers and disconnectors).

For each element of the voltage level, we also provide the bus breaker bus where it is connected.

Parameters

- **network_handle** (*pypowsybl._pypowsybl.JavaHandle*)
- **voltage_level_id** (*str*)

create_graph()

Representation of the topology as a networkx graph.

Return type
Graph

property buses: DataFrame

The list of buses of the bus breaker view, as a dataframe.

The dataframe includes the following columns:

- **name**: Name of the bus breaker view bus
- **bus_id**: id of the corresponding bus in the bus view.

This dataframe is indexed by the id of the bus breaker view bus.

property elements: DataFrame

The list of elements (lines, generators...) of this voltage level, together with the bus of the bus breaker view where they are connected.

The dataframe includes the following columns:

- **type**: Type of the connected element (GENERATOR, LINE, ...)
- **bus_id**: bus id of the bus breaker view
- **side**: Side of the connected element

This dataframe is indexed by the id of the connected elements.

property switches: DataFrame

The list of switches of the bus breaker view, together with their connection status, as a dataframe.

The dataframe includes the following columns:

- **kind**: Switch kind (BREAKER, DISCONNECTOR, ...)
- **open**: True if the switch is opened
- **bus1_id**: node where the switch is connected at side 1
- **bus2_id**: node where the switch is connected at side 2

This dataframe is indexed by the id of the switches.

pypowsybl.network.NodeBreakerTopology

class NodeBreakerTopology(*network_handle*, *voltage_level_id*)

Node-breaker representation of the topology of a voltage level.

The topology is actually represented as a graph, where vertices are called "nodes" and are identified by a unique number in the voltage level, while edges are switches (breakers and disconnectors), or internal connections (plain "wires").

Parameters

- **network_handle** (*pypowsybl._pypowsybl.JavaHandle*)
- **voltage_level_id** (*str*)

create_graph()

Representation of the topology as a networkx graph.

Return type
Graph

property internal_connections: DataFrame

The list of internal connection of the voltage level, together with the nodes they connect.

The dataframe includes the following columns:

- **node1**: node where the internal connection is connected at side 1
- **node2**: node where the internal connection is connected at side 2

property nodes: DataFrame

The list of nodes of the voltage level, together with their corresponding network element (if any), as a dataframe.

The dataframe includes the following columns:

- **connectable_id**: Connected element ID, if any.
- **connectable_type**: Connected element type, if any.

This dataframe is indexed by the id of the nodes.

property switches: DataFrame

The list of switches of the voltage level, together with their connection status, as a dataframe.

The dataframe includes the following columns:

- **name**: Switch name
- **kind**: Switch kind (BREAKER, DISCONNECTOR, ...)
- **open**: True if the switch is opened
- **retained**: True if the switch is to be retained in the Bus/Breaker view
- **node1**: node where the switch is connected at side 1
- **node2**: node where the switch is connected at side 2

This dataframe is indexed by the id of the switches.

Network elements update

Network elements can be modified using dataframes:

Network.update_2_windings_transformers	Update 2 windings transformers with data provided as a DataFrame or as named arguments.
Network.update_3_windings_transformers	Update 3 windings transformers with data provided as a DataFrame or as named arguments.
Network.update_areas	Update areas with data provided as a DataFrame or as named arguments.
Network.update_batteries	Update batteries with data provided as a DataFrame or as named arguments.
Network.update_branches	Update branches with data provided as a DataFrame or as named arguments.
Network.update_buses	Update buses with data provided as a dataframe or as named arguments.
Network.update_busbar_sections	Update bus bar sections with a Pandas dataframe.
Network.update_boundary_lines	Update boundary lines with data provided as a DataFrame or as named arguments.
Network.update_dangling_lines	
Network.update_boundary_lines_generation	Update boundary lines generation part with data provided as a DataFrame or as named arguments.
Network.update_dangling_lines_generation	
Network.update_generators	Update generators with data provided as a DataFrame or as named arguments.

continues on next page

Table 7 – continued from previous page

Network.update_grounds	Update grounds with data provided as a DataFrame or as named arguments.
Network.update_hvdc_lines	Update HVDC lines with data provided as a DataFrame or as named arguments.
Network.update_injections	Update injections with data provided as a DataFrame or as named arguments.
Network.update_lcc_converter_stations	Update VSC converter stations with data provided as a DataFrame or as named arguments.
Network.update_linear_shunt_compensator_sect	Update shunt compensators with data provided as a DataFrame or as named arguments.
Network.update_lines	Update lines data with data provided as a DataFrame or as named arguments.
Network.update_loads	Update loads with data provided as a DataFrame or as named arguments.
*Network.update_non_linear_shunt_compensator_*	Update non linear shunt compensators sections with data provided as a DataFrame or as named arguments.
Network.update_operational_limits	Update operational limits values with data provided as a DataFrame or as named arguments.
Network.update_phase_tap_changers	Update phase tap changers with data provided as a DataFrame or as named arguments.
Network.update_phase_tap_changer_steps	Update phase tap changer steps with data provided as a DataFrame or as named arguments.
Network.update_ratio_tap_changers	Update ratio tap changers with data provided as a DataFrame or as named arguments.
Network.update_ratio_tap_changer_steps	Update ratio tap changer steps with data provided as a DataFrame or as named arguments.
Network.update_shunt_compensators	Update shunt compensators with data provided as a DataFrame or as named arguments.
Network.update_static_var_compensators	Update static var compensators with data provided as a DataFrame or as named arguments.
Network.update_substations	Update substations with data provided as a DataFrame or as named arguments.
Network.update_switches	Update switches with data provided as a DataFrame or as named arguments.
Network.update_terminals	Update terminals with data provided as a DataFrame or as named arguments.
Network.update_tie_lines	Update tie lines with data provided as a DataFrame or as named arguments.
Network.update_voltage_levels	Update voltage levels with data provided as a DataFrame or as named arguments.
Network.update_vsc_converter_stations	Update VSC converter stations with data provided as a DataFrame or as named arguments.
Network.update_dc_nodes	Update dc nodes with data provided as a dataframe or as named arguments.
Network.update_dc_lines	Update dc lines data with data provided as a DataFrame or as named arguments.
Network.update_voltage_source_converters	Update voltage source converters with data provided as a DataFrame or as named arguments.
Network.update_dc_grounds	Update dc grounds with data provided as a dataframe or as named arguments.
Network.update_dc_buses	Update dc buses with data provided as a dataframe or as named arguments.

continues on next page

Table 7 – continued from previous page

Network.get_elements_properties	Get a dataframe of properties of all network elements.
Network.add_elements_properties	Add properties to network elements, provided as a DataFrame or as named arguments.
Network.remove_elements_properties	Remove properties from a list of network elements
Network.add_aliases	Adds aliases to network elements.
Network.remove_aliases	Removes aliases of network elements.
Network.apply_solved_values	Replaces the "input" values used for load flow calculation by their solved values returned by the load flow calculation.
Network.apply_solved_tap_position_and_sectio	Replaces the "input" values used for load flow calculation by their solved values returned by the load flow calculation, only for tap position and section count.

pypowsybl.network.Network.update_2_windings_transformers

Network.**update_2_windings_transformers**(*df=None*, ***kwargs*)

Update 2 windings transformers with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *r*
- *x*
- *g*
- *b*
- *rated_u1*
- *rated_u2*
- *rated_s*
- *p1*
- *q1*
- *p2*
- *q2*
- *connected1*
- *connected2*

- *fictitious*
- *selected_limits_group_1*
- *selected_limits_group_2*

See also

get_2_windings_transformers()

Examples

Some examples using keyword arguments:

```
network.update_2_windings_transformers(id='T-1', connected1=False, connected2=False)
network.update_2_windings_transformers(id=['T-1', 'T-2'], r=[0.5, 2.0], x=[5, 10])
```

pypowsybl.network.Network.update_3_windings_transformers

Network.**update_3_windings_transformers**(*df=None*, ***kwargs*)

Update 3 windings transformers with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *r1*
- *x1*
- *g1*
- *b1*
- *rated_u1*
- *rated_s1*
- *p1*
- *q1*
- *connected1*
- *ratio_tap_position1*
- *phase_tap_position1*
- *selected_limits_group_1*

- *r2*
- *x2*
- *g2*
- *b2*
- *rated_u2*
- *rated_s2*
- *p2*
- *q2*
- *connected2*
- *ratio_tap_position2*
- *phase_tap_position2*
- *selected_limits_group_2*
- *r3*
- *x3*
- *g3*
- *b3*
- *rated_u3*
- *rated_s3*
- *p3*
- *q3*
- *connected3*
- *ratio_tap_position3*
- *phase_tap_position3*
- *selected_limits_group_3*
- *fictitious*

See also

get_3_windings_transformers()

Examples

Some examples using keyword arguments:

```
network.update_3_windings_transformers(id='T-1', connected1=False, connected2=False,
↪ connected3=False)
network.update_3_windings_transformers(id=['T-1', 'T-2'], r3=[0.5, 2.0], x3=[5, 10])
```

pypowsybl.network.Network.update_areas

Network.**update_areas**(*df=None*, ***kwargs*)

Update areas with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *interchange_target*
- *fictitious*

See also

get_areas()

Examples

Some examples using keyword arguments:

```
network.update_areas(id='ControlArea_A', interchange_target=-500)
network.update_areas(id=['ControlArea_A', 'ControlArea_B'], interchange_target=[-
↪500, 500])
```

pypowsybl.network.Network.update_batteries

Network.**update_batteries**(*df=None*, ***kwargs*)

Update batteries with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *target_p*
- *target_q*
- *connected*
- *max_q*
- *min_q*
- *fictitious*

See also

get_batteries()

Examples

Some examples using keyword arguments:

```
network.update_batteries(id='B-1', p0=10, q0=3)
network.update_batteries(id=['B-1', 'B-2'],  p0=[10, 20], q0=[3, 5])
```

pypowsybl.network.Network.update_branches

Network.**update_branches**(*df=None*, ***kwargs*)

Update branches with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are :

- *connected1*
- *connected2*
- *bus_breaker_bus1_id2*
- *bus_breaker_bus2_id2*

See also

get_branches()

Examples

Some examples using keyword arguments:

```
network.update_branches(element_id='BRANCH_ID', connected1=False, connected2=False)
```

pypowsybl.network.Network.update_buses

Network.**update_buses**(*df=None*, ***kwargs*)

Update buses with data provided as a dataframe or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *v_mag*
- *v_angle*
- *fictitious*

See also

get_buses()

Examples

Some examples using keyword arguments:

```
network.update_buses(id='B1', v_mag=400.0)
network.update_buses(id=['B1', 'B2'], v_mag=[400.0, 63.5])
```

pypowsybl.network.Network.update_busbar_sections

Network.**update_busbar_sections**(*df=None*, ***kwargs*)

Update bus bar sections with a Pandas dataframe.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named

arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

pypowsybl.network.Network.update_boundary_lines

Network.**update_boundary_lines**(*df=None*, ***kwargs*)

Update boundary lines with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *r*
- *x*
- *g*
- *b*
- *p0*
- *q0*
- *p*
- *q*
- *connected*
- *fictitious*
- *pairing_key*
- *bus_breaker_bus_id* if the boundary line is in a voltage level with *BUS_BREAKER* topology
- *selected_limits_group*

See also

get_boundary_lines()

Examples

Some examples using keyword arguments:

```
network.update_boundary_lines(id='L-1', p0=10, q0=3)
network.update_boundary_lines(id=['L-1', 'L-2'],  p0=[10, 20], q0=[3, 5])
```

pypowsybl.network.Network.update_dangling_lines

Network.**update_dangling_lines**(*df=None*, ***kwargs*)

Deprecated since version 1.15.0: Use *update_boundary_lines()* instead.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type

None

pypowsybl.network.Network.update_boundary_lines_generation

Network.**update_boundary_lines_generation**(*df=None*, ***kwargs*)

Update boundary lines generation part with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *min_p*
- *max_p*
- *target_p*
- *target_q*
- *target_v*
- *voltage_regulator_on*

See also

get_boundary_lines_generation()

Examples

Some examples using keyword arguments:

```
network.update_boundary_lines_generation(id='BL', voltage_regulator_on=True, target_
↪v=225)
network.update_boundary_lines_generation(id=['BL', 'BL2'],  voltage_regulator_
↪on=[True, True], target_v=[225, 400])
```

pypowsybl.network.Network.update_dangling_lines_generation

Network.**update_dangling_lines_generation**(*df=None*, ***kwargs*)

Deprecated since version 1.15.0: Use *update_boundary_lines_generation()* instead.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type

None

pypowsybl.network.Network.update_generators

Network.**update_generators**(*df=None*, ***kwargs*)

Update generators with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *target_p*
- *max_p*
- *min_p*
- *rated_s*
- *target_v*
- *target_q*
- *voltage_regulator_on*

- ***regulated_element_id*: you may define any injection or busbar section as the regulated location.**
  Only supported in node breaker voltage levels.
- *p*
- *q*
- *connected*
- *fictitious*

See also

get_generators()

Examples

Some examples using keyword arguments:

```
network.update_generators(id='G-1', connected=True, target_p=500)
network.update_generators(id=['G-1', 'G-2'], target_v=[403, 401])
```

pypowsybl.network.Network.update_grounds

`Network.update_grounds`(*df=None*, ***kwargs*)

Update grounds with data provided as a `DataFrame` or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *connected*

See also

get_grounds()

Examples

Some examples using keyword arguments:

```
network.update_grounds(id='L-1', connected=False)
```

pypowsybl.network.Network.update_hvdc_lines

Network.**update_hvdc_lines**(*df=None*, ***kwargs*)

Update HVDC lines with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *converters_mode*
- *target_p*
- *max_p*
- *nominal_v*
- *r*
- *connected1*
- *connected2*
- *fictitious*

See also

get_hvdc_lines()

Examples

Some examples using keyword arguments:

```
network.update_hvdc_lines(id='HVDC-1', target_p=800)
network.update_hvdc_lines(id=['HVDC-1', 'HVDC-2'], target_p=[800, 600])
```

pypowsybl.network.Network.update_injections

Network.**update_injections**(*df=None*, ***kwargs*)

Update injections with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str*

| *_NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are :

- *connected*
- *bus_breaker_bus_id*

See also

get_injections()

Examples

Some examples using keyword arguments:

```
network.update_injections(element_id='INJECTION_ID', connected=True, bus_breaker_
↪bus_id='B2')
```

pypowsybl.network.Network.update_lcc_converter_stations

`Network.update_lcc_converter_stations`(*df=None*, ***kwargs*)

Update VSC converter stations with data provided as a `DataFrame` or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *power_factor*
- *loss_factor*
- *p*
- *q*
- *connected*
- *fictitious*

See also

get_vsc_converter_stations()

Examples

Some examples using keyword arguments:

```
network.update_lcc_converter_stations(id='S-1', connected=True)
network.update_lcc_converter_stations(id=['S-1', 'S-2'], connected=[True, False])
```

pypowsybl.network.Network.update_linear_shunt_compensator_sections

Network.**update_linear_shunt_compensator_sections**(*df=None*, ***kwargs*)

Update shunt compensators with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *g_per_section*
- *b_per_section*
- *max_section_count*

See also

get_linear_shunt_compensator_sections()

Examples

Some examples using keyword arguments:

```
network.update_linear_shunt_compensator_sections(id='CAP-1', max_section_count=3)
```

pypowsybl.network.Network.update_lines

Network.**update_lines**(*df=None*, ***kwargs*)

Update lines data with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *r*
- *x*
- *g1*
- *b1*
- *g2*
- *b2*
- *p1*
- *q1*
- *p2*
- *q2*
- *connected1*
- *connected2*
- *fictitious*
- *selected_limits_group_1*
- *selected_limits_group_2*

See also

get_lines()

Examples

Some examples using keyword arguments:

```
network.update_lines(id='L-1', connected1=False, connected2=True)
network.update_lines(id=['L-1', 'L-2'], r=[0.5, 2.0], x=[5, 10])
```

pypowsybl.network.Network.update_loads

Network.**update_loads**(*df=None*, ***kwargs*)

Update loads with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *p0*
- *q0*
- *connected*
- *fictitious*

See also

get_loads()

Examples

Some examples using keyword arguments:

```
network.update_loads(id='L-1', p0=10, q0=3)
network.update_loads(id=['L-1', 'L-2'],  p0=[10, 20], q0=[3, 5])
```

pypowsybl.network.Network.update_non_linear_shunt_compensator_sections

Network.**update_non_linear_shunt_compensator_sections**(*df=None*, ***kwargs*)

Update non linear shunt compensators sections with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are :

- *g*
- *b*

See also

get_non_linear_shunt_compensator_sections()

Examples

Some examples using keyword arguments:

```
network.update_non_linear_shunt_compensator_sections(id='CAP-1', section=1, b=1e-5)
```

pypowsybl.network.Network.update_operational_limits

Network.**update_operational_limits**(*df=None*, ***kwargs*)

Update operational limits values with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Only the value of operational limits can be updated. To define which limit must be modified, the following fields must be present :

- *element_id*
- *side*
- *type*
- *acceptable_duration*
- *group_name* (if not specified, will try to update the corresponding limit in the selected set of the element)

See also

get_operational_limits()

Examples

An example using keyword arguments:

```
network.update_operational_limits(id='LINE', side='ONE', type='CURRENT', acceptable_
↪duration=600, value=500)
```

pypowsybl.network.Network.update_phase_tap_changers

Network.**update_phase_tap_changers**(*df=None*, ***kwargs*)

Update phase tap changers with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated :

- *tap*
- *regulating*
- *regulation_mode*
- *regulation_value*
- *regulated_side*
- *target_deadband*
- *fictitious*

See also

get_phase_tap_changers()

Examples

Some examples using keyword arguments:

```
network.update_phase_tap_changers(id='T-1', regulation_mode=CURRENT_LIMITER,␣
↪regulation_value=500)
network.update_phase_tap_changers(id=['T-1', 'T-2'], tap=[12, 25])
```

pypowsybl.network.Network.update_phase_tap_changer_steps

Network.**update_phase_tap_changer_steps**(*df=None*, ***kwargs*)

Update phase tap changer steps with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named

arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated :

- *rho*
- *alpha*
- *r*
- *x*
- *g*
- *b*

See also

get_phase_tap_changer_steps()

Examples

Some examples using keyword arguments:

```
network.update_phase_tap_changer_steps(id='T-1', position=2, rho=1.1, alpha=-12.3)
```

pypowsybl.network.Network.update_ratio_tap_changers

Network.**update_ratio_tap_changers**(*df=None*, ***kwargs*)

Update ratio tap changers with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *tap*
- *oltc*
- *regulating*,
- *regulated_side*,

- *target_v*
- *target_deadband*

See also

get_ratio_tap_changers()

Examples

Some examples using keyword arguments:

```
network.update_ratio_tap_changers(id='T-1', tap=12)
network.update_ratio_tap_changers(id=['T-1', 'T-2'], target_v=[64, 65],␣
↪regulating=[True, True])
```

pypowsybl.network.Network.update_ratio_tap_changer_steps

Network.**update_ratio_tap_changer_steps**(*df=None*, ***kwargs*)

Update ratio tap changer steps with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *rho*
- *r*
- *x*
- *g*
- *b*

See also

get_ratio_tap_changer_steps()

Examples

Some examples using keyword arguments:

```
network.update_ratio_tap_changer_steps(id='T-1', position=2, rho=1.1)
```

pypowsybl.network.Network.update_shunt_compensators

Network.**update_shunt_compensators**(*df=None*, ***kwargs*)

Update shunt compensators with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *section_count*
- *p*
- *q*
- *connected*
- *fictitious*

See also

get_shunt_compensators()

Examples

Some examples using keyword arguments:

```
network.update_shunt_compensators(id='IND-1', section_count=1, connected=True)
network.update_shunt_compensators(id=['IND-1', 'CAP-1'], section_count=[1, 0])
```

pypowsybl.network.Network.update_static_var_compensators

Network.**update_static_var_compensators**(*df=None*, ***kwargs*)

Update static var compensators with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named

arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *b_min*
- *b_max*
- *target_v*
- *target_q*
- *regulation_mode*
- *p*
- *q*
- *connected*
- *regulated_element_id*
- *fictitious*

See also

get_static_var_compensators()

Examples

Some examples using keyword arguments:

```
network.update_static_var_compensators(id='SVC-1', target_v=225)
network.update_static_var_compensators(id=['SVC-1', 'SVC-2'], target_v=[226, 405])
```

pypowsybl.network.Network.update_substations

Network.**update_substations**(*df=None*, ***kwargs*)

Update substations with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are :

- *TSO*
- *country*
- *fictitious*

See also

get_substations()

Examples

Some examples using keyword arguments:

```
network.update_substations(id='S-1', TSO='ELIA', country='BE')
network.update_substations(id=['S-1', 'S-2'], country=['BE', 'FR'])
```

pypowsybl.network.Network.update_switches

Network.**update_switches**(*df=None*, ***kwargs*)

Update switches with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *open*
- *retained*
- *fictitious*

See also

get_switches()

Examples

Some examples using keyword arguments:

```
network.update_switches(id='BREAKER-1', open=True)
network.update_switches(id=['BREAKER-1', 'DISC-2'], open=[True, False])
```

pypowsybl.network.Network.update_terminals

Network.**update_terminals**(*df=None*, ***kwargs*)

Update terminals with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are :

- *connected*: element_side must be provided if it is a sided network element

See also

get_terminals()

Examples

Some examples using keyword arguments:

```
network.update_terminals(element_id='GENERATOR_ID', connected=False)
network.update_terminals(element_id='LINE_ID', element_side='ONE', connected=True)
```

pypowsybl.network.Network.update_tie_lines

Network.**update_tie_lines**(*df=None*, ***kwargs*)

Update tie lines with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are :

- *fictitious*
- *connected1*
- *connected2*

See also

get_tie_lines()

Examples

Some examples using keyword arguments:

```
network.update_tie_lines(element_id='TIE_LINE_ID', fictitious=True)
```

pypowsybl.network.Network.update_voltage_levels

Network.**update_voltage_levels**(*df=None*, ***kwargs*)

Update voltage levels with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are :

- *high_voltage_limit*
- *low_voltage_limit*
- *nominal_v*
- *fictitious*

See also

get_voltage_levels()

Examples

Some examples using keyword arguments:

```
network.update_voltage_levels(id='VL-1', high_voltage_limit=420)
network.update_voltage_levels(id=['VL-1', 'VL-2'], low_voltage_limit=[385, 390])
```

pypowsybl.network.Network.update_vsc_converter_stations

Network.**update_vsc_converter_stations**(*df=None*, ***kwargs*)

Update VSC converter stations with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *loss_factor*
- *target_v*
- *target_q*
- *voltage_regulator_on*
- *p*
- *q*
- *connected*
- *regulated_element_id*
- *fictitious*

See also

get_vsc_converter_stations()

Examples

Some examples using keyword arguments:

```
network.update_vsc_converter_stations(id='S-1', target_v=400, voltage_regulator_
↪on=True)
network.update_vsc_converter_stations(id=['S-1', 'S-2'], target_v=[400, 400])
```

pypowsybl.network.Network.update_dc_nodes

Network.**update_dc_nodes**(*df=None*, ***kwargs*)

Update dc nodes with data provided as a dataframe or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *v*
- *fictitious*

See also

get_dc_nodes()

Examples

Some examples using keyword arguments:

```
network.update_dc_nodes(id='DN1', v=400.0)
network.update_dc_nodes(id=['DN1', 'DN2'], v=[400.0, 63.5])
```

pypowsybl.network.Network.update_dc_lines

Network.**update_dc_lines**(*df=None*, ***kwargs*)

Update dc lines data with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named

arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *r*
- *i1*
- *i2*
- *fictitious*

See also

get_dc_lines()

Examples

Some examples using keyword arguments:

```
network.update_dc_lines(id='L-1', i1 = 1.2)
network.update_dc_lines(id=['L-1', 'L-2'], r=[0.5, 2.0])
```

pypowsybl.network.Network.update_voltage_source_converters

Network.**update_voltage_source_converters**(*df=None*, ***kwargs*)

Update voltage source converters with data provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *target_v_dc*
- *target_v_ac*
- *target_p*
- *target_q*
- *p_ac*

- *q_ac*
- *p_dc1*
- *p_dc2*
- *fictitious*

See also

get_voltage_source_converters()

Examples

Some examples using keyword arguments:

```
network.update_voltage_source_converters(id='CONV-1', p=40)
network.update_voltage_source_converters(id=['CONV-1', 'CONV-2'], target_p=[40, 40])
```

pypowsybl.network.Network.update_dc_grounds

Network.**update_dc_grounds**(*df=None*, ***kwargs*)

Update dc grounds with data provided as a dataframe or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type

None

Notes

Attributes that can be updated are:

- *r*
- *fictitious*

See also

get_dc_grounds()

Examples

Some examples using keyword arguments:

```
network.update_dc_grounds(id='DG1', r=1.0)
network.update_dc_grounds(id=['DG1', 'DG2'], r=[2.0, 0.0])
```

pypowsybl.network.Network.update_dc_buses

Network.**update_dc_buses**(*df=None*, ***kwargs*)

Update dc buses with data provided as a dataframe or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the data to be updated, as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

Attributes that can be updated are:

- *v*
- *fictitious*

See also

get_dc_buses()

Examples

Some examples using keyword arguments:

```
network.update_dc_buses(id='DB1', v=400.0)
network.update_dc_buses(id=['DB1', 'DB2'], v=[400.0, 63.5])
```

pypowsybl.network.Network.get_elements_properties

Network.**get_elements_properties**(*all_attributes=False*, *attributes=None*, ***kwargs*)

Get a dataframe of properties of all network elements.

Args:

Returns
A dataframe of properties

Parameters

- **all_attributes** (*bool*)
- **attributes** (*List[str] | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
DataFrame

Notes

The resulting dataframe, depending on the parameters, will include the following columns:

- **type**: the type of the network element (network, line, generator, load, ...)
- **key**: property key
- **value**: property value

This dataframe is indexed on the network element ID.

pypowsybl.network.Network.add_elements_properties

Network.**add_elements_properties**(*df=None*, ***kwargs*)

Add properties to network elements, provided as a DataFrame or as named arguments.

Parameters

- **df** (*DataFrame | None*) – the properties to be created or updated. The index has to be the *id* identifying the network elements.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the properties to be added as named arguments. Arguments can be a single string or any type of sequence of strings. In the case of sequences, all arguments must have the same length.

Return type

None

Examples

For example, to add the properties prop1 = value1 and prop2 = value2 to a network element:

```
>>> network.add_elements_properties(id='GENERATOR-1', prop1='value1', prop2='value2
↪')
>>> network.get_generators(attributes=['prop1', 'prop2'], id='GENERATOR-1')
              prop1  prop2
id
GENERATOR-1  value1 value2
```

You can also update multiple elements at once, for example with a dataframe:

```
>>> properties_df = pd.Dataframe(index=pd.Series('id', ['G1', 'G2']),
                                 data={
                                     'prop1': [ 'val11', 'val12'],
                                     'prop2': [ 'val12', 'val22'],
                                 })
>>> network.add_elements_properties(properties_df)
>>> network.get_generators(attributes=['prop1', 'prop2'], id=['G1', 'G2'])
         prop1  prop2
id
G1       val11  val12
G2       val21  val22
```

pypowsybl.network.Network.remove_elements_properties

Network.**remove_elements_properties**(*ids*, *properties*)

Remove properties from a list of network elements

Parameters

- **ids** (*str | List[str]*) – list of the network elements that will have their properties removed
- **properties** (*str | List[str]*) – list of the properties that will be removed

Return type
None

Examples

To remove properties prop1 and prop2 from network elements GEN1 and GEN2:

```
network.remove_elements_properties(ids=['GEN1', 'GEN2'], properties=['prop1', 'prop2
↪'])
```

pypowsybl.network.Network.add_aliases

Network.**add_aliases**(*df=None*, ***kwargs*)

Adds aliases to network elements.

An alias is a reference to a network element. For example, to get or to update an element, his alias may be used instead of his id. An alias may be associated with a type, to distinguish it from other aliases.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the network element associated to the alias
- **alias**: name of the alias
- **alias_type**: type of the alias (optional)

Examples

```
network.add_aliases(id='element_id', alias='alias_id')
network.add_aliases(id='element_id', alias='alias_id', alias_type='alias_type')
```

pypowsybl.network.Network.remove_aliases

Network.**remove_aliases**(*df=None*, ***kwargs*)

Removes aliases of network elements.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the network element associated to the alias
- **alias**: name of the alias

Examples

```
network.remove_aliases(id='element_id', alias='alias_id')
```

pypowsybl.network.Network.apply_solved_values

Network.**apply_solved_values**()

Replaces the "input" values used for load flow calculation by their solved values returned by the load flow calculation. The copied values are : - **solved_tap_position** -> **tap** for ratio/phase tap changers - **solved_section_count** -> **section_count** for shunt/compensators - **targetP/Q** -> **P/Q** for generators, boundary lines, batteries - **targetV** -> **V** for generators and boundary lines - **P0/Q0** -> **P/Q** for loads

Return type
None

pypowsybl.network.Network.apply_solved_tap_position_and_section_count_values

Network.**apply_solved_tap_position_and_section_count_values**()

Replaces the "input" values used for load flow calculation by their solved values returned by the load flow calculation, only for tap position and section count. The copied values are : - **solved_tap_position** -> **tap** for ratio/phase tap changers - **solved_section_count** -> **section_count** for shunt/compensators

Return type
None

Network elements creation and removal

Network elements can be created or removed using the following methods:

Network.create_2_windings_transformers	Creates 2 windings transformers.
Network.create_3_windings_transformers	Creates three-winding transformers.
Network.create_areas	Create areas.
Network.create_areas_voltage_levels	Associate voltage levels to (existing) areas.
Network.create_areas_boundaries	Define boundaries of (existing) areas.
Network.create_batteries	Creates batteries.
Network.create_busbar_sections	Creates bus bar sections.
Network.create_buses	Creates buses in bus-breaker voltage levels.
Network.create_curve_reactive_limits	Creates reactive limits as "curves".
Network.create_boundary_lines	Creates boundary lines.
Network.create_dangling_lines	
Network.create_generators	Creates generators.
Network.create_grounds	Create grounds.
Network.create_hvdc_lines	Creates HVDC lines.
Network.create_internal_connections	Creates internal connections.
Network.create_lcc_converter_stations	Creates LCC converter stations.
Network.create_lines	Creates lines.
Network.create_loads	Create loads.
Network.create_minmax_reactive_limits	Creates reactive limits of type min/max.
Network.create_operational_limits	Creates operational limits.
Network.create_phase_tap_changers	Create phase tap changers on transformers.
Network.create_ratio_tap_changers	Create ratio tap changers on transformers.
Network.create_shunt_compensators	Create shunt compensators.
Network.create_static_var_compensators	Creates static var compensators.
Network.create_substations	Creates substations.
Network.create_switches	Creates switches.
Network.create_voltage_levels	Creates voltage levels.
Network.create_vsc_converter_stations	Creates VSC converter stations.
Network.create_tie_lines	Creates tie lines from two boundary lines.
Network.create_dc_nodes	Creates DC nodes.
Network.create_dc_lines	Creates DC lines.
Network.create_voltage_source_converters	Creates voltage source converter.
Network.create_dc_grounds	Creates DC grounds.
Network.remove_elements	Removes elements from the network.
Network.remove_internal_connections	Removes internal connections.

pypowsybl.network.Network.create_2_windings_transformers

Network.**create_2_windings_transformers**(*df=None*, ***kwargs*)

Creates 2 windings transformers.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new transformer
- **voltage_level1_id**: the voltage level where the new transformer will be connected on side 1. The voltage level must already exist.
- **bus1_id**: the bus where the new transformer will be connected on side 1, if the voltage level has a bus-breaker topology kind.
- **connectable_bus1_id**: the bus where the new transformer will be connectable on side 1, if the voltage level has a bus-breaker topology kind.
- **node1**: the node where the new transformer will be connected on side 1, if the voltage level has a node-breaker topology kind.
- **voltage_level2_id**: the voltage level where the new transformer will be connected on side 2. The voltage level must already exist.
- **bus2_id**: the bus where the new transformer will be connected on side 2, if the voltage level has a bus-breaker topology kind.
- **connectable_bus2_id**: the bus where the new transformer will be connectable on side 2, if the voltage level has a bus-breaker topology kind.
- **node2**: the node where the new transformer will be connected on side 2, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **rated_u1**: nominal voltage of the side 1 of the transformer
- **rated_u2**: nominal voltage of the side 2 of the transformer
- **rated_s**: nominal power of the transformer
- **b**: the shunt susceptance, in S, on side 2
- **g**: the shunt conductance, in S, on side 2
- **r**: the resistance, in Ohm, on side 2
- **x**: the reactance, in Ohm, on side 2

Examples

Using keyword arguments:

```
network.create_2_windings_transformers(id='T-1', voltage_level1_id='VL1', bus1_id=
↪'B1',
                                       voltage_level2_id='VL2', bus2_id='B2',
                                       b=1e-6, g=1e-6, r=0.5, x=10, rated_u1=400,␣
↪rated_u2=225)
```

pypowsybl.network.Network.create_3_windings_transformers

Network.**create_3_windings_transformers**(*df=None*, ***kwargs*)

Creates three-winding transformers.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

For each side of the transformer, either a node ID (voltage level in node-breaker topology), bus ID (voltage level in bus-breaker topology) or connectable bus ID (voltage level in bus-breaker topology, the transformer is created associated but disconnected from this bus) should be specified.

Valid attributes are:

- **id**: the identifier of the new transformer
- **rated_u0**: the rated voltage at the star bus
- **voltage_level1_id**: the voltage level where the new transformer will be connected on side 1. The voltage level must already exist.
- **bus1_id**: the bus where the new transformer will be connected on side 1, if the voltage level has a bus-breaker topology kind.
- **connectable_bus1_id**: the bus to which the transformer can be connected on side 1, if the voltage level has a bus-breaker topology kind. The transformer is created disconnected from this bus.
- **node1**: the node where the new transformer will be connected on side 1, if the voltage level has a node-breaker topology kind.
- **voltage_level2_id**: the voltage level where the new transformer will be connected on side 2. The voltage level must already exist.
- **bus2_id**: the bus where the new transformer will be connected on side 2, if the voltage level has a bus-breaker topology kind.
- **connectable_bus2_id**: the bus to which the transformer can be connected on side 2, if the voltage level has a bus-breaker topology kind. The transformer is created disconnected from this bus.
- **node2**: the node where the new transformer will be connected on side 2, if the voltage level has a node-breaker topology kind.
- **voltage_level3_id**: the voltage level where the new transformer will be connected on side 3. The voltage level must already exist.
- **bus3_id**: the bus where the new transformer will be connected on side 3, if the voltage level has a bus-breaker topology kind.
- **connectable_bus3_id**: the bus to which the transformer can be connected on side 3, if the voltage level has a bus-breaker topology kind. The transformer is created disconnected from this bus.

- **node3**: the node where the new transformer will be connected on side 3, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **rated_u1**: nominal voltage of the side 1 of the transformer
- **rated_u2**: nominal voltage of the side 2 of the transformer
- **rated_u3**: nominal voltage of the side 3 of the transformer
- **rated_s1**: optionally, nominal power of the side 1 of the transformer
- **rated_s2**: optionally, nominal power of the side 2 of the transformer
- **rated_s3**: optionally, nominal power of the side 3 of the transformer
- **r1**: the resistance of the side 1, in Ohm
- **r2**: the resistance of the side 2, in Ohm
- **r3**: the resistance of the side 3, in Ohm
- **x1**: the reactance of the side 1, in Ohm
- **x2**: the reactance of the side 2, in Ohm
- **x3**: the reactance of the side 3, in Ohm
- **b1**: the shunt susceptance of the side 1, in S
- **b2**: the shunt susceptance of the side 2, in S
- **b3**: the shunt susceptance of the side 3, in S
- **g1**: the shunt conductance of the side 1, in S
- **g2**: the shunt conductance of the side 2, in S
- **g3**: the shunt conductance of the side 3, in S

Examples

Using keyword arguments:

```
network.create_3_windings_transformers(id='T-1', rated_u0 = 225, voltage_level1_id=
↪'VL1', bus1_id='B1',
                                       voltage_level2_id='VL2', bus2_id='B2',
                                       voltage_level3_id='VL3', bus3_id='B3',
                                       b1=1e-6, g1=1e-6, r1=0.5, x1=10, rated_
↪u1=400, rated_s1=100,
                                       b2=1e-6, g2=1e-6, r2=0.5, x2=10, rated_
↪u2=225, rated_s2=100,
                                       b3=1e-6, g3=1e-6, r3=0.5, x3=10, rated_u3=90,
↪ rated_s3=100)
```

pypowsybl.network.Network.create_areas

Network.**create_areas**(*df=None*, ***kwargs*)

Create areas.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

See also

get_areas()

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new area
- **name**: an optional human-readable name
- **area_type**: the type of Area (e.g. ControlArea, BiddingZone ...)
- **interchange_target**: Target active power interchange (MW)

Examples

Using keyword arguments:

```
network.create_areas(id='Area1', area_type='ControlArea', interchange_target=120.5)
```

pypowsybl.network.Network.create_areas_voltage_levels

Network.**create_areas_voltage_levels**(*df=None*, ***kwargs*)

Associate voltage levels to (existing) areas.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Important: The provided voltage levels for an area replace all existing voltage levels of that area, i.e. the entire list of voltage levels must be provided for the areas being edited.

Valid attributes are:

- **id**: the identifier of the area

- **voltage_level_id**: the identifier of the voltage level to be associated with the area

See also

get_areas_voltage_levels()

Examples

To associate voltage levels VL1 and VL2 to Area1.

```
network.create_areas_voltage_levels(id=['Area1', 'Area1'], voltage_level_id=['VL1',
↪'VL2'])
```

To dissociate all VoltageLevels of a given area, provide an empty string in voltage_level_id.

```
network.create_areas_voltage_levels(id=['Area1'], voltage_level_id=[''])
```

pypowsybl.network.Network.create_areas_boundaries

Network.**create_areas_boundaries**(*df=None*, ***kwargs*)

Define boundaries of (existing) areas.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Important: The provided boundaries for an area replace all existing boundaries of that area, i.e. the entire list of boundaries must be provided for the areas being edited.

Valid attributes are:

- **id**: the identifier of the area
- **boundary_type**: either *BOUNDARY_LINE* or *TERMINAL*, defaults to *BOUNDARY_LINE*.
- **element**: boundary line identifier, or any connectable
- **side**: if element is not a boundary line (e.g. a branch or transformer), the terminal side
- **ac**: True is boundary is to be considered as AC

See also

get_areas_boundaries()

Examples

```
# define boundary lines NHV1_XNODE1 and NVH1_XNODE2 as boundaries of AreaA, and
# define boundary lines XNODE1_NHV2 and XNODE2_NHV2 as boundaries of AreaB
network.create_areas_boundaries(id=['AreaA', 'AreaA', 'AreaB', 'AreaB'],
                                boundary_type=['BOUNDARY_LINE', 'BOUNDARY_LINE',
↪'BOUNDARY_LINE', 'BOUNDARY_LINE'],
                                element=['NHV1_XNODE1', 'NVH1_XNODE2', 'XNODE1_NHV2
↪', 'XNODE2_NHV2'],
                                ac=[True, True, True, True])
```

To dissociate all Boundaries of a given area, provide an empty string in element.

```
network.create_areas_boundaries(id=['Area1'], element=[''])
```

pypowsybl.network.Network.create_batteries

Network.**create_batteries**(*df=None*, ***kwargs*)

Creates batteries.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new battery
- **voltage_level_id**: the voltage level where the new battery will be created. The voltage level must already exist.
- **bus_id**: the bus where the new battery will be connected, if the voltage level has a bus-breaker topology kind.
- **connectable_bus_id**: the bus where the new battery will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new battery will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **min_p**: minimum active power, in MW
- **max_p**: maximum active power, in MW
- **target_p**: active power consumption, in MW
- **target_q**: reactive power consumption, in MVar

Examples

Using keyword arguments:

```
network.create_batteries(id='BAT-1', voltage_level_id='VL1', bus_id='B1',
                         min_p=5, max_p=50, p0=10, q0=3)
```

pypowsybl.network.Network.create_busbar_sections

Network.**create_busbar_sections**(*df=None*, ***kwargs*)

Creates bus bar sections.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new busbar section
- **voltage_level_id**: the voltage level where the new busbar section will be created. The voltage level must already exist.
- **node**: the node where the new generator will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name

Examples

Using keyword arguments:

```
network.create_busbar_sections(id='BBS', voltage_level_id='VL1', node=0)
```

pypowsybl.network.Network.create_buses

Network.**create_buses**(*df=None*, ***kwargs*)

Creates buses in bus-breaker voltage levels.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

This method can only create "configured buses", in bus-breaker voltage levels, as opposed to electrical buses computed from this underlying topology.

Valid attributes are:

- **id**: the identifier of the new configured bus
- **voltage_level_id**: the voltage level where the new bus will be created. The voltage level must already exist, and must have a bus-breaker topology kind.
- **name**: an optional human-readable name

Examples

Using keyword arguments:

```
network.create_buses(id='B1', voltage_level_id='VL1')
```

pypowsybl.network.Network.create_curve_reactive_limits

Network.**create_curve_reactive_limits**(*df=None*, ***kwargs*)

Creates reactive limits as "curves".

Curves are actually composed of line segments, defined by a list of points. Each row of the input data actually defines 2 points: one for the minimum limit, one for the maximum limit, for the given active power value.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the generator
- **p**: active power, in MW, for which this row defines limits
- **min_q**: minimum reactive limit at this active power value, in MVAr
- **max_q**: maximum reactive limit at this active power value, in MVAr

At least 2 rows must be defined for each generator, for 2 different active power values. Previously defined limits for a given generator, if present, will be replaced by the new ones.

Examples

Generator GEN-1 will be able to provide 150MVAr when P=0MW, and only 100MVAr when it generates 100MW:

```
network.create_curve_reactive_limits(id=['GEN-1', 'GEN-1'],
                                     p=[0, 100],
                                     min_q=[-150, -100],
                                     max_q=[150, 100])
```

See also

create_minmax_reactive_limits()

pypowsybl.network.Network.create_boundary_lines

Network.**create_boundary_lines**(*df=None*, *generation_df=Empty DataFrame Columns: [] Index: []*, ***kwargs*)

Creates boundary lines.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **generation_df** (*DataFrame*) – Attributes of the boundary lines optional generation part, only as a dataframe
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

General boundary line data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new line
- **voltage_level_id**: the voltage level where the new line will be created. The voltage level must already exist.
- **bus_id**: the bus where the new line will be connected, if the voltage level has a bus-breaker topology kind.
- **connectable_bus_id**: the bus where the new line will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new line will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **p0**: the active power consumption, in MW
- **q0**: the reactive power consumption, in MVar
- **r**: the resistance, in Ohms
- **x**: the reactance, in Ohms

- **g**: the shunt conductance, in S
- **b**: the shunt susceptance, in S
- **pairing_key**: the optional pairing key associated to the boundary line, to be used for creating tie lines.
- **ucte_xnode_code**: deprecated, use pairing_key instead.

Boundary line generation information must be provided as a dataframe. Valid attributes are:

- **id**: Identifier of the boundary line that contains this generation part
- **min_p**: Minimum active power output of the boundary line's generation part
- **max_p**: Maximum active power output of the boundary line's generation part
- **target_p**: Active power target of the generation part
- **target_q**: Reactive power target of the generation part
- **target_v**: Voltage target of the generation part
- **voltage_regulator_on**: `True` if the generation part regulates voltage

Examples

Using keyword arguments:

```
network.create_boundary_lines(id='BAT-1', voltage_level_id='VL1', bus_id='B1',
                              p0=10, q0=3, r=0, x=5, g=0, b=1e-6)
```

pypowsybl.network.Network.create_dangling_lines

Network.**create_dangling_lines**(*df=None*, *generation_df=Empty DataFrame Columns: [] Index: []*, ***kwargs*)

Deprecated since version 1.15.0: Use *create_boundary_lines()* instead.

Parameters

- **df** (*DataFrame | None*)
- **generation_df** (*DataFrame*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

pypowsybl.network.Network.create_generators

Network.**create_generators**(*df=None*, ***kwargs*)

Creates generators.

Parameters

- **df** (*DataFrame | None*) – Attributes as dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new generator
- **voltage_level_id**: the voltage level where the new generator will be created. The voltage level must already exist.
- **bus_id**: the bus where the new generator will be connected, if the voltage level has a bus-breaker topology kind.
- **connectable_bus_id**: the bus where the new generator will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new generator will be connected, if the voltage level has a node-breaker topology kind.
- **energy_source**: the type of energy source (HYDRO, NUCLEAR, ...)
- **condenser**: define if the generator is a condenser (boolean)
- **max_p**: maximum active power in MW
- **min_p**: minimum active power in MW
- **target_p**: target active power in MW
- **target_q**: target reactive power in MVar, when the generator does not regulate voltage
- **rated_s**: nominal power in MVA
- **target_v**: target voltage in kV, when the generator regulates voltage
- **equivalent_local_target_v**: local target voltage in kV, equivalent to the potential remote **target_v** (that must be set to use this)
- **voltage_regulator_on**: true if the generator regulates voltage

Examples

Using keyword arguments:

```
network.create_generators(id='GEN',
                          voltage_level_id='VL1',
                          bus_id='B1',
                          target_p=100,
                          min_p=0,
                          max_p=200,
                          target_v=400,
                          voltage_regulator_on=True)
```

pypowsybl.network.Network.create_grounds

Network.**create_grounds**(*df=None*, ***kwargs*)

Create grounds.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new ground
- **voltage_level_id**: the voltage level where the new ground will be created. The voltage level must already exist.
- **bus_id**: the bus where the new ground will be connected, if the voltage level has a bus-breaker topology kind.
- **connectable_bus_id**: the bus where the new ground will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new ground will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name

Examples

Using keyword arguments:

```
network.create_loads(id='GROUND-1', voltage_level_id='VL1', bus_id='B1')
```

pypowsybl.network.Network.create_hvdc_lines

Network.**create_hvdc_lines**(*df=None*, ***kwargs*)

Creates HVDC lines.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new HVDC line
- **name**: an optional human-readable name
- **converter_station1_id**: the station where the new HVDC line will be connected on side 1. It must already exist.
- **converter_station2_id**: the station where the new HVDC line will be connected on side 2. It must already exist.
- **r**: the resistance of the HVDC line, in Ohm
- **nominal_v**: the nominal voltage of the HVDC line, in kV
- **max_p**: the maximum transmissible power, in MW
- **target_p**: the active power target, in MW
- **converters_mode**: SIDE_1_RECTIFIER_SIDE_2_INVERTER or SIDE_1_INVERTER_SIDE_2_RECTIFIER

Examples

Using keyword arguments:

```
network.create_hvdc_lines(id='HVDC-1', converter_station1_id='CS-1', converter_
↪station2_id='CS-2',
                          r=1.0, nominal_v=400, converters_mode='SIDE_1_RECTIFIER_
↪SIDE_2_INVERTER',
                          max_p=1000, target_p=800)
```

pypowsybl.network.Network.create_internal_connections

Network.**create_internal_connections**(*df=None*, ***kwargs*)

Creates internal connections.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

See also

- *get_node_breaker_topology()*
- *remove_internal_connections()*

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **voltage_level_id**: voltage level identifier. The voltage level must be in Node/Breaker topology kind.
- **node1**: node 1 of the internal connection
- **node2**: node 2 of the internal connection

Examples

Using keyword arguments:

```
network.create_internal_connections(voltage_level_id='VL1', node1=3, node2=6)
```

pypowsybl.network.Network.create_lcc_converter_stations

Network.**create_lcc_converter_stations**(*df=None*, ***kwargs*)

Creates LCC converter stations.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new station
- **voltage_level_id**: the voltage level where the new station will be created. The voltage level must already exist.
- **bus_id**: the bus where the new station will be connected, if the voltage level has a bus-breaker topology kind.
- **connectable_bus_id**: the bus where the new station will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new station will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **power_factor**: the power factor (ratio of the active power to the apparent power)
- **loss_factor**: the loss factor of the station

Examples

Using keyword arguments:

```
network.create_lcc_converter_stations(id='CS-1', voltage_level_id='VL1', bus_id='B1
↪',
                                      power_factor=0.3, loss_factor=0.1)
```

pypowsybl.network.Network.create_lines

Network.**create_lines**(*df=None*, ***kwargs*)

Creates lines.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new line
- **voltage_level1_id**: the voltage level where the new line will be connected on side 1. The voltage level must already exist.
- **bus1_id**: the bus where the new line will be connected on side 1, if the voltage level has a bus-breaker topology kind.
- **connectable_bus1_id**: the bus where the new line will be connectable on side 1, if the voltage level has a bus-breaker topology kind.
- **node1**: the node where the new line will be connected on side 1, if the voltage level has a node-breaker topology kind.
- **voltage_level2_id**: the voltage level where the new line will be connected on side 2. The voltage level must already exist.
- **bus2_id**: the bus where the new line will be connected on side 2, if the voltage level has a bus-breaker topology kind.
- **connectable_bus2_id**: the bus where the new line will be connectable on side 2, if the voltage level has a bus-breaker topology kind.
- **node2**: the node where the new line will be connected on side 2, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **b1**: the shunt susceptance, in S, on side 1
- **b2**: the shunt susceptance, in S, on side 2

- **g1**: the shunt conductance, in S, on side 1
- **g2**: the shunt conductance, in S, on side 2
- **r**: the resistance, in Ohm
- **x**: the reactance, in Ohm

Examples

Using keyword arguments:

```
network.create_lines(id='LINE-1', voltage_level1_id='VL1', bus1_id='B1',
                     voltage_level2_id='VL2', bus2_id='B2',
                     b1=1e-6, b2=1e-6, g1=0, g2=0, r=0.5, x=10)
```

pypowsybl.network.Network.create_loads

Network.**create_loads**(*df=None*, ***kwargs*)

Create loads.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new load
- **voltage_level_id**: the voltage level where the new load will be created. The voltage level must already exist.
- **bus_id**: the bus where the new load will be connected, if the voltage level has a bus-breaker topology kind.
- **connectable_bus_id**: the bus where the new load will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new load will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **type**: optionally, the type of load (UNDEFINED, AUXILIARY, FICTITIOUS)
- **p0**: active power load, in MW
- **q0**: reactive power load, in MVar

Examples

Using keyword arguments:

```
network.create_loads(id='LOAD-1', voltage_level_id='VL1', bus_id='B1', p0=10, q0=3)
```

pypowsybl.network.Network.create_minmax_reactive_limits

Network.**create_minmax_reactive_limits**(*df=None*, ***kwargs*)

Creates reactive limits of type min/max.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the generator
- **min_q**: minimum reactive limit, in MVAr
- **max_q**: maximum reactive limit, in MVAr

Previously defined limits for a given generator, if present, will be replaced by the new ones.

Examples

Using keyword arguments:

```
network.create_minmax_reactive_limits(id='GEN-1', min_q=-100, max_q=100)
```

See also

create_curve_reactive_limits()

pypowsybl.network.Network.create_operational_limits

Network.**create_operational_limits**(*df=None*, ***kwargs*)

Creates operational limits.

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **element_id**: the ID of the network element on which we want to create new limits
- **side**: the side of the network element where we want to create new limits (ONE, TWO, THREE)
- **name**: the name of the limit
- **type**: the type of limit to be created (CURRENT, APPARENT_POWER, ACTIVE_POWER)
- **value**: the value of the limit in A, MVA or MW
- **acceptable_duration**: the maximum number of seconds during which we can operate under that limit
- **fictitious**: fictitious limit ?
- **group_name**: Name of the operational limit group to add the limit to (if not specified, the limit is added to the group "DEFAULT")

For each location of the network defined by a couple (element_id, side) and each group_name:

- if operational limits already exist, they will be replaced
- multiple limits may be defined, typically with different acceptable_duration
- you can only define ONE permanent limit, identified by an acceptable_duration of -1

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

pypowsybl.network.Network.create_phase_tap_changers

Network.**create_phase_tap_changers**(*ptc_df*, *steps_df*)

Create phase tap changers on transformers.

Tap changers data must be provided in 2 separate dataframes: one for the tap changers attributes, and another one for tap changers steps attributes. The latter one will generally have multiple lines for one transformer ID. The steps are created in order of the dataframe order meaning (for a transformer) first line of the steps dataframe will create step one of the steps of these transformer.

Parameters

- **ptc_df** (*DataFrame*) – dataframe of tap changers data
- **steps_df** (*DataFrame*) – dataframe of steps data

Return type
None

Notes

Valid attributes for the tap changers dataframe are:

- **id**: the transformer where this tap changer will be created
- **tap**: the current tap position
- **low_tap**: the number of the lowest tap position (default 0)

- **regulation_mode**: the regulation mode (CURRENT_LIMITER or ACTIVE_POWER_CONTROL)
- **target_deadband**: the regulation deadband
- **regulating**: true if the tap changer should regulate
- **regulated_side**: the side where the current or active power is regulated (ONE or TWO if two-winding transformer, ONE, TWO or THREE if three-winding transformer)
- **side**: Side of the tap changer (only for three-winding transformers)

Valid attributes for the steps dataframe are:

- **id**: the transformer where this step will be added
- **g**: the shunt conductance increase compared to the transformer, for this step, in percentage
- **b**: the shunt susceptance increase compared to the transformer, for this step, in percentage
- **r**: the resistance increase compared to the transformer, for this step, in percentage
- **x**: the reactance increased compared to the transformer, for this step, in percentage
- **rho**: the transformer ratio for this step (1 means real ratio is rated_u2/rated_u1)
- **alpha**: the phase shift, in degrees, for this step

Examples

We need to provide 2 dataframes, 1 for tap changer basic data, and one for step-wise data:

```
ptc_df = pd.DataFrame.from_records(
    index='id', columns=['id', 'target_deadband', 'regulation_mode', 'low_tap', 'tap
↪'],
    data=[('TWT_TEST', 2, 'CURRENT_LIMITER', 0, 1)])
steps_df = pd.DataFrame.from_records(
    index='id', columns=['id', 'b', 'g', 'r', 'x', 'rho', 'alpha'],
    data=[('TWT_TEST', 2, 2, 1, 1, 0.5, 0.1),
          ('TWT_TEST', 2, 2, 1, 1, 0.4, 0.2),
          ('TWT_TEST', 2, 2, 1, 1, 0.5, 0.1)])
n.create_phase_tap_changers(ptc_df, steps_df)
```

pypowsybl.network.Network.create_ratio_tap_changers

Network.**create_ratio_tap_changers**(*rtc_df*, *steps_df*)

Create ratio tap changers on transformers.

Tap changers data must be provided in 2 separate dataframes: one for the tap changers attributes, and another one for tap changers steps attributes. The latter one will generally have multiple lines for one transformer ID. The steps are created in order of the dataframe order meaning (for a transformer) first line of the steps dataframe will create step one of the steps of these transformer.

Parameters

- **rtc_df** (*DataFrame*) – dataframe of tap changers data
- **steps_df** (*DataFrame*) – dataframe of steps data

Return type

None

Notes

Valid attributes for the tap changers dataframe are:

- **id**: the transformer where this tap changer will be created
- **tap**: the current tap position
- **low_tap**: the number of the lowest tap position (default 0)
- **oltc**: true if the transformer has on-load voltage regulation capability
- **target_v**: the target voltage, in kV
- **target_deadband**: the target voltage regulation deadband, in kV
- **regulating**: true if the tap changer should regulate voltage (**oltc** must be true to set this to true)
- **regulated_side**: the side where voltage is regulated (ONE or TWO if two-winding transformer, ONE, TWO or THREE if three-winding transformer)
- **side**: Side of the tap changer (only for three-winding transformers)

Valid attributes for the steps dataframe are:

- **id**: the transformer where this step will be added
- **g**: the shunt conductance increase compared to the transformer, for this step, in percentage
- **b**: the shunt susceptance increase compared to the transformer, for this step, in percentage
- **r**: the resistance increase compared to the transformer, for this step, in percentage
- **x**: the reactance increased compared to the transformer, for this step, in percentage
- **rho**: the transformer ratio for this step (1 means real ratio is rated_u2/rated_u1)

Examples

We need to provide 2 dataframes, 1 for tap changer basic data, and one for step-wise data:

```
rtc_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'target_deadband', 'target_v', 'oltc', 'low_tap', 'tap'],
    data=[('NGEN_NHV1', 2, 200, False, 0, 1)])
steps_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'b', 'g', 'r', 'x', 'rho'],
    data=[('NGEN_NHV1', 2, 2, 1, 1, 0.5),
          ('NGEN_NHV1', 2, 2, 1, 1, 0.5),
          ('NGEN_NHV1', 2, 2, 1, 1, 0.8)])
network.create_ratio_tap_changers(rtc_df, steps_df)
```

pypowsybl.network.Network.create_shunt_compensators

Network.**create_shunt_compensators**(*shunt_df*, *linear_model_df=None*, *non_linear_model_df=None*)

Create shunt compensators.

Shunt compensator sections can be described in 1 of 2 ways: either with a linear model, with a maximum section count and a per-section values, or with a non linear model, where each section is described individually.

For this reason, 2 or 3 dataframes need to be provided: one for shunt compensators data, optionally one for linear models, and optionally one for non linear models.

Parameters

- **shunt_df** (*DataFrame*) – dataframe for shunt compensators data
- **linear_model_df** (*DataFrame | None*) – dataframe for linear model sections data
- **non_linear_model_df** (*DataFrame | None*) – dataframe for sections data

Return type
None

Notes

Valid attributes for the shunt compensators dataframe are:

- **id**: the identifier of the new shunt
- **voltage_level_id**: the voltage level where the new shunt will be created. The voltage level must already exist.
- **bus_id**: the bus where the new shunt will be connected, if the voltage level has a bus-breaker topology kind.
- **connectable_bus_id**: the bus where the new shunt will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new shunt will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **model_type**: either LINEAR or NON_LINEAR
- **section_count**: the current count of connected sections
- **target_v**: an optional target voltage in kV
- **target_v**: an optional deadband for the target voltage, in kV

Valid attributes for the linear sections models are:

- **id**: the identifier of the new shunt
- **g_per_section**: the conductance, in Ohm, for each section
- **b_per_section**: the susceptance, in Ohm, for each section
- **max_section_count**: the maximum number of connectable sections

This dataframe must have only one row for each shunt compensator.

Valid attributes for the non linear sections models are:

- **id**: the identifier of the new shunt
- **g**: the conductance, in Ohm, for this section
- **b**: the susceptance, in Ohm, for this section

This dataframe will have multiple rows for each shunt compensator: one by section.

Examples

For example, to create linear model shunts, we need 1 dataframe for the shunts and 1 dataframe for the linear model of sections:

```
shunt_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'name', 'model_type', 'section_count', 'target_v',
             'target_deadband', 'voltage_level_id', 'node'],
    data=[('SHUNT-1', '', 'LINEAR', 1, 400, 2, 'S1VL2', 2)])
model_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'g_per_section', 'b_per_section', 'max_section_count'],
    data=[('SHUNT-1', 0.14, -0.01, 2)])
n.create_shunt_compensators(shunt_df, model_df)
```

For non linear model shunts, we need 1 dataframe for the shunts and 1 dataframe for the sections:

```
shunt_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'name', 'model_type', 'section_count', 'target_v',
             'target_deadband', 'voltage_level_id', 'node'],
    data=[('SHUNT1', '', 'NON_LINEAR', 1, 400, 2, 'S1VL2', 2),
          ('SHUNT2', '', 'NON_LINEAR', 1, 400, 2, 'S1VL2', 10)])
model_df = pd.DataFrame.from_records(
    index='id',
    columns=['id', 'g', 'b'],
    data=[('SHUNT1', 1, 2),
          ('SHUNT1', 3, 4),
          ('SHUNT2', 5, 6),
          ('SHUNT2', 7, 8)])
n.create_shunt_compensators(shunt_df, non_linear_model_df=model_df)
```

pypowsybl.network.Network.create_static_var_compensators

Network.**create_static_var_compensators**(*df=None*, ***kwargs*)

Creates static var compensators.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new SVC
- **voltage_level_id**: the voltage level where the new SVC will be created. The voltage level must already exist.
- **bus_id**: the bus where the new SVC will be connected, if the voltage level has a bus-breaker topology kind.

- **connectable_bus_id**: the bus where the new SVC will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new SVC will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **b_max**: the maximum susceptance, in S
- **b_min**: the minimum susceptance, in S
- **regulation_mode**: the regulation mode (VOLTAGE, REACTIVE_POWER)
- **regulating**: True if the regulation is active
- **target_v**: the target voltage, in kV, when the regulation mode is VOLTAGE
- **target_q**: the target reactive power, in MVar, when the regulation mode is not VOLTAGE

Examples

Using keyword arguments:

```
network.create_static_var_compensators(id='CS-1', voltage_level_id='VL1', bus_id='B1
↪',
                                       b_min=-0.01, b_max=0.01, regulation_mode=
↪'VOLTAGE',
                                       regulating=True, target_v=400.0)
```

pypowsybl.network.Network.create_substations

Network.**create_substations**(*df=None*, ***kwargs*)

Creates substations.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are: - **id**: the identifier of the substation - **name**: an optional human readable name for the substation - **country**: an optional country code ('DE', 'IT', ...) - **TSO**: an optional TSO name

Examples

Using keyword arguments:

```
network.create_substations(id='S-1', country='IT', TSO='TERNA')
```

Or using a dataframe:

```
stations = pd.DataFrame.from_records(index='id', data=[
    {'id': 'S1', 'country': 'BE'},
    {'id': 'S2', 'country': 'DE'}
])
network.create_substations(stations)
```

pypowsybl.network.Network.create_switches

Network.**create_switches**(*df=None*, ***kwargs*)

Creates switches.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new switch
- **voltage_level_id**: the voltage level where the new switch will be connected. The voltage level must already exist.
- **bus1_id**: the bus where the new switch will be connected on side 1, if the voltage level has a bus-breaker topology kind.
- **bus2_id**: the bus where the new switch will be connected on side 2, if the voltage level has a bus-breaker topology kind.
- **node1**: the node where the new switch will be connected on side 1, if the voltage level has a node-breaker topology kind.
- **node2**: the node where the new switch will be connected on side 2, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **kind**: the kind of switch (BREAKER, DISCONNECTOR, LOAD_BREAK_SWITCH)
- **open**: true if the switch is open, default false
- **retained**: true if the switch should be retained in bus-breaker topology, default false
- **fictitious**: true if the switch is fictitious, default false

Examples

Using keyword arguments:

```
# In a bus-breaker voltage level, between configured buses B1 and B2
network.create_switches(id='BREAKER-1', voltage_level_id='VL1', bus1_id='B1', bus2_
↪id='B2',
                        kind='BREAKER', open=False)

# In a node-breaker voltage level, between nodes 5 and 7
network.create_switches(id='BREAKER-1', voltage_level_id='VL1', node1=5, node2=7,
                        kind='BREAKER', open=False)
```

pypowsybl.network.Network.create_voltage_levels

Network.**create_voltage_levels**(*df=None*, ***kwargs*)

Creates voltage levels.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new voltage level
- **substation_id**: the identifier of the substation which the new voltage level belongs to. Optional. If defined, the substation must already exist.
- **name**: an optional human-readable name
- **topology_kind**: the topology kind, BUS_BREAKER or NODE_BREAKER
- **nominal_v**: the nominal voltage, in kV
- **low_voltage_limit**: the lower operational voltage limit, in kV
- **high_voltage_limit**: the upper operational voltage limit, in kV

Examples

Using keyword arguments:

```
network.create_voltage_levels(id='VL1', substation_id='S1', topology_kind='BUS_
↪BREAKER',
                              nominal_v=400, low_voltage_limit=380, high_voltage_
↪limit=420)
```

pypowsybl.network.Network.create_vsc_converter_stations

Network.**create_vsc_converter_stations**(*df=None*, ***kwargs*)

Creates VSC converter stations.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new station
- **voltage_level_id**: the voltage level where the new station will be created. The voltage level must already exist.
- **bus_id**: the bus where the new station will be connected, if the voltage level has a bus-breaker topology kind.
- **connectable_bus_id**: the bus where the new station will be connectable, if the voltage level has a bus-breaker topology kind.
- **node**: the node where the new station will be connected, if the voltage level has a node-breaker topology kind.
- **name**: an optional human-readable name
- **loss_factor**: the loss factor of the new station
- **voltage_regulator_on**: true if the station regulated voltage
- **target_v**: the target voltage, in kV, when the station regulates voltage
- **target_q**: the target reactive power, in MVar, when the station does not regulate voltage

Examples

Using keyword arguments:

```
network.create_vsc_converter_stations(id='CS-1', voltage_level_id='VL1', bus_id='B1
↪',
                                      loss_factor=0.1, voltage_regulator_on=True,␣
↪target_v=400.0)
```

pypowsybl.network.Network.create_tie_lines

Network.**create_tie_lines**(*df=None*, ***kwargs*)

Creates tie lines from two boundary lines. Both boundary lines must have the same pairing key (formerly named UCTE Xnode code).

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new tie line
- **name**: an optional human-readable name
- **boundary_line1_id**: the ID of the first boundary line It must already exist.
- **boundary_line2_id**: the ID of the second boundary line It must already exist.

dangling_line1_id and **dangling_line2_id** are deprecated attributes, use **boundary_line1_id** and **boundary_line2_id** instead.

Examples

Using keyword arguments:

```
network.create_tie_lines(id='tie_line_1', boundary_line1_id='BL-1', boundary_line2_
↪id='BL-2')
```

pypowsybl.network.Network.create_dc_nodes

Network.**create_dc_nodes**(*df=None*, ***kwargs*)

Creates DC nodes.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new DC node
- **name**: an optional human-readable name

- **nominal_v**: the nominal voltage of the DC node

Examples

Using keyword arguments:

```
network.create_dc_nodes(id='DN1', nominal_v=400.0)
```

pypowsybl.network.Network.create_dc_lines

Network.**create_dc_lines**(*df=None*, ***kwargs*)

Creates DC lines.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new DC line
- **name**: an optional human-readable name
- **dc_node1_id**: the DC node where the new DC line will be connected on side 1. It must already exist.
- **dc_node2_id**: the DC node where the new DC line will be connected on side 2. It must already exist.
- **r**: the resistance of the DC line, in Ohm

Examples

Using keyword arguments:

```
network.create_dc_lines(id='DL1', dc_node1_id='DN1', dc_node2_id='DN2', r=1.0)
```

pypowsybl.network.Network.create_voltage_source_converters

Network.**create_voltage_source_converters**(*df=None*, ***kwargs*)

Creates voltage source converter.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new voltage source converter
- **name**: an optional human-readable name
- **voltage_level_id**: the voltage level where the new converter will be connected. The voltage level must already exist.
- **bus_id1 the bus where the new converter will be connected on side 1.**
  It must already exist.
- **bus_id2 the bus where the new converter will be connected on side 2.**
  It must already exist.
- **dc_node1_id**: the DC node where the new converter will be connected on DC side 1. It must already exist.
- **dc_node2_id**: the DC node where the new converter will be connected on DC side 2. It must already exist.
- **dc_connected1**: defines if the converter is connected at the dc node 1 (boolean)
- **dc_connected2**: defines if the converter is connected at the dc node 2 (boolean)
- **regulating_element_id: the network PCC element**
  It must already exist.
- **voltage_regulator_on**: defines if the converter regulates AC voltage (boolean)
- **control_mode** the control mode of the converter (V_DC or P_PCC)
- **target_p** the AC active power setpoint
- **target_q** the AC reactive power setpoint
- **target_v_ac** the AC voltage setpoint
- **target_v_dc** the DC voltage setpoint
- **idle_loss** the idle loss coefficient
- **switching_loss** the switching loss coefficient
- **resistive_loss** the resistive loss coefficient

Examples

Using keyword arguments:

```
network.create_voltage_source_converter(id='VSC-1', voltage_level_id='VL1',dc_node1_
↪id='DN1',
dc_node2_id='DN2', bus1_id='B1', voltage_regulator_on=0, control_mode='P_PCC',␣
↪target_p=50.0,
target_q=50.0, target_v_ac=300.0, target_v_dc=400.0, idle_loss=1.0, switching_
↪loss=2.0,
resistive_loss=3.0)
```

pypowsybl.network.Network.create_dc_grounds

Network.**create_dc_grounds**(*df=None*, ***kwargs*)

Creates DC grounds.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the identifier of the new DC ground
- **name**: an optional human-readable name
- **dc_node_id** the id of the DC Node connected to the DC ground
- **r**: the resistance of the DC ground

Examples

Using keyword arguments:

```
network.create_dc_grounds(id='DG1', dc_node_id="DN1", r=1.0)
```

pypowsybl.network.Network.remove_elements

Network.**remove_elements**(*elements_ids*)

Removes elements from the network.

Parameters
elements_ids (*str | List[str]*) – IDs of the elements to be removed.

Return type
None

Notes

Elements can be provided as a list of IDs or as a single ID.

Elements can be any identifiable object of the network (line, generator, switch, substation ...).

Examples

```
network.remove_elements('GENERATOR-1')  # Removes only 1 element
network.remove_elements(['GENERATOR-1', 'BUS'])
```

pypowsybl.network.Network.remove_internal_connections

Network.**remove_internal_connections**(*df=None*, ***kwargs*)

Removes internal connections.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

See also

- *get_node_breaker_topology()*
- *create_internal_connections()*

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **voltage_level_id**: voltage level identifier. The voltage level must be in Node/Breaker topology kind.
- **node1**: node 1 of the internal connection
- **node2**: node 2 of the internal connection

Examples

Using keyword arguments:

```
network.remove_internal_connections(voltage_level_id='VL1', node1=3, node2=6)
```

Network variants management

Network variants may be used to manage multiple states of the network efficiently.

Network.get_working_variant_id	The current working variant ID.
Network.clone_variant	Creates a copy of the source variant
Network.set_working_variant	Changes the working variant.
Network.remove_variant	Removes a variant from the network.
Network.get_variant_ids	Get the list of existing variant IDs.

pypowsybl.network.Network.get_working_variant_id

Network.**get_working_variant_id**()

The current working variant ID.

Returns
the id of the currently selected variant.

Return type
str

pypowsybl.network.Network.clone_variant

Network.**clone_variant**(*src*, *target*, *may_overwrite=True*)

Creates a copy of the source variant

Parameters

- **src** (*str*) – variant to copy
- **target** (*str*) – id of the new variant that will be a copy of src
- **may_overwrite** (*bool*) – indicates if the target can be overwritten when it already exists

Return type
None

pypowsybl.network.Network.set_working_variant

Network.**set_working_variant**(*variant*)

Changes the working variant. The provided variant ID must correspond to an existing variant, for example created by a call to *clone_variant*.

Parameters
variant (*str*) – id of the variant selected (it must exist)

Return type
None

pypowsybl.network.Network.remove_variant

Network.**remove_variant**(*variant*)

Removes a variant from the network.

Parameters
variant (*str*) – id of the variant to be deleted

Return type
None

pypowsybl.network.Network.get_variant_ids

Network.**get_variant_ids**()

Get the list of existing variant IDs.

Returns
all the ids of the existing variants

Return type
List[str]

Network elements extensions

get_extensions_names	Get the list of available extensions.
get_extensions_information	Get more information about extensions
Network.get_extensions	Get an extension as a DataFrame for a specified extension name.
Network.create_extensions	create extensions of network elements with data provided as a DataFrame.
Network.update_extensions	Update extensions of network elements with data provided as a DataFrame.
Network.remove_extensions	Removes network elements extensions, given the extension's name.

pypowsybl.network.get_extensions_names

get_extensions_names()

Get the list of available extensions.

Returns
the names of the available extensions

Return type
List[str]

pypowsybl.network.get_extensions_information

get_extensions_information()

Get more information about extensions

Returns
a dataframe with information about extensions

Return type
DataFrame

pypowsybl.network.Network.get_extensions

Network.**get_extensions**(*extension_name*, *table_name=''*)

Get an extension as a DataFrame for a specified extension name.

Parameters

- **extension_name** (*str*) – name of the extension
- **table_name** (*str*) – optional argument to choose the name of the dataframe to retrieve for extensions using multiple dataframes

Returns
A dataframe with the extensions data.

Return type
DataFrame

Notes

The extra id column in the resulting dataframe provides the link to the extensions parent elements

pypowsybl.network.Network.create_extensions

Network.**create_extensions**(*extension_name*, *df=None*, ***kwargs*)

create extensions of network elements with data provided as a DataFrame.

Parameters

- **extension_name** (*str*) – name of the extension
- **df** (*DataFrame | List[DataFrame | None] | None*) – the data to be created A single dataframe or a list of dataframes can be given as arguments
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be created, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

The id column in the dataframe provides the link to the extensions parent elements

pypowsybl.network.Network.update_extensions

Network.**update_extensions**(*extension_name*, *df=None*, *table_name=''*, ***kwargs*)

Update extensions of network elements with data provided as a DataFrame.

Parameters

- **extension_name** (*str*) – name of the extension
- **table_name** (*str*) – for multiple dataframes extensions, to precise which dataframe to modify
- **df** (*DataFrame | None*) – the data to be updated
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be updated, as named arguments. Arguments can be single values or any type of sequence. In the case of sequences, all arguments must have the same length.

Return type
None

Notes

The id column in the dataframe provides the link to the extensions parent elements

pypowsybl.network.Network.remove_extensions

Network.**remove_extensions**(*extension_name*, *ids*)

Removes network elements extensions, given the extension's name.

Parameters

- **extension_name** (*str*) – name of the extension
- **ids** (*str | List[str]*) – IDs of the elements to be removed.

Return type
None

Notes

ids can be provided as a list of IDs or as a single ID.

Miscellaneous network functions

Network.reduce	
Network.reduce_by_ids	Reduce to a smaller network (only keeping voltage levels whose id is in the specified list)
Network.reduce_by_voltage_range	Reduce to a smaller network (only keeping all elements whose nominal voltage is in the specified voltage range)
Network.reduce_by_ids_and_depths	Reduce to a smaller network (keeping the specified voltage levels with all respective neighbours at most at the specified depth).
Network.merge	Merges networks into this one.
Network.get_single_line_diagram	Create a single line diagram from a voltage level or a substation.
Network.write_single_line_diagram_svg	Create a single line diagram in SVG format from a voltage level or a substation and write to a file.
Network.get_matrix_multi_substation_single_l	Create a single line diagram from multiple substations
Network.write_matrix_multi_substation_single	Create a single line diagram in SVG format from a voltage level or a substation and write to a file.
Network.get_network_area_diagram	Create a network area diagram.
Network.write_network_area_diagram_svg	
Network.write_network_area_diagram	Create a network area diagram in SVG format and write it to a file.
Network.get_network_area_diagram_displayed_v	Gathers the name of the displayed voltage levels of a network-area diagram in a list, according to the input voltage level(s) and the depth of the diagram.
Network.get_default_nad_profile	Creates a default NadProfile for branch labels, twt labels, injection labels, bus descriptions, and VL descriptions.
Network.disconnect	Disconnects a connectable element's terminal or terminals.
Network.connect	Connects a connectable element's terminal or terminals.
Network.open_switch	
Network.close_switch	
Network.get_validation_level	The network's validation level.
Network.validate	Validate the network.
Network.set_min_validation_level	Set the minimum validation level for the network.

pypowsybl.network.Network.reduce

Network.**reduce**(*v_min=0*, *v_max=1.7976931348623157e+308*, *ids=None*, *vl_depths=None*, *with_dangling_lines=False*)

Deprecated since version 1.14.0: Use *reduce_by_voltage_range()*, *reduce_by_ids()* or *reduce_by_ids_and_depths()* instead depending on your use case.

Reduce to a smaller network according to the following parameters

Parameters

- **v_min** (*float*) – minimum voltage of the voltage levels kept after reducing
- **v_max** (*float*) – voltage maximum of the voltage levels kept after reducing
- **ids** (*List[str] | None*) – ids of the voltage levels that will be kept
- **vl_depths** (*List[tuple] | None*) – depth around voltage levels which are indicated by their id, that will be kept
- **with_dangling_lines** (*bool*) – keeping the dangling lines

Return type
None

pypowsybl.network.Network.reduce_by_ids

Network.**reduce_by_ids**(*ids*, *with_boundary_lines=False*, *with_dangling_lines=None*)

Reduce to a smaller network (only keeping voltage levels whose id is in the specified list)

Parameters

- **ids** (*List[str]*) – list of the voltage level ids that should be kept in the reduced network
- **with_boundary_lines** (*bool*) – whether boundary lines should be created to replace lines cut at the boundary of reduction
- **with_dangling_lines** (*bool | None*) – deprecated, use with_boundary_lines instead

Return type
None

Example

```
network.reduce_by_ids(ids=["VL1", "VL2"])
```

will only keep voltage levels VL1 and VL2 and all network elements between them.

pypowsybl.network.Network.reduce_by_voltage_range

Network.**reduce_by_voltage_range**(*v_min=0*, *v_max=1.7976931348623157e+308*, *with_boundary_lines=False*, *with_dangling_lines=None*)

Reduce to a smaller network (only keeping all elements whose nominal voltage is in the specified voltage range)

Parameters

- **v_min** (*float*) – minimum voltage of the voltage levels kept after reducing
- **v_max** (*float*) – voltage maximum of the voltage levels kept after reducing
- **with_boundary_lines** (*bool*) – whether boundary lines should be created to replace lines cut at the boundary of reduction

- **with_dangling_lines** (*bool | None*) – deprecated, use with_boundary_lines instead

Return type
None

Example

```
network.reduce_by_voltage_range(v_min=90, v_max=250, with_boundary_lines=True)
```

will only keep elements of voltage level between 90 and 250kV, replacing the lines cut at the boundary by boundary lines.

pypowsybl.network.Network.reduce_by_ids_and_depths

Network.**reduce_by_ids_and_depths**(*vl_depths*, *with_boundary_lines=False*, *with_dangling_lines=None*)

Reduce to a smaller network (keeping the specified voltage levels with all respective neighbours at most at the specified depth).

Parameters

- **vl_depths** (*List[tuple[str, int]]*) – list of the voltage level ids that should be kept in the reduced network
- **with_boundary_lines** (*bool*) – whether boundary lines should be created to replace lines cut at the boundary of reduction
- **with_dangling_lines** (*bool | None*) – deprecated, use with_boundary_lines instead

Return type
None

Example

```
network.reduce_by_ids_and_depths(vl_depths=[("VL1", 1), ("VL25", 3)])
```

will only keep voltage levels VL1 and its neighbours, and VL25 with all elements around it with at most 3 connections between them.

pypowsybl.network.Network.merge

Network.**merge**(*networks*)

Merges networks into this one.

Parameters
networks (Network | *Sequence[Network]*) – List of networks to be merged into this one.

Return type
None

Examples

If you have 3 networks, you can merge this way:

```
network1.merge([network2, network3])
```

Note that network1 is modified: it absorbs network2 and network3.

pypowsybl.network.Network.get_single_line_diagram

Network.**get_single_line_diagram**(*container_id*, *parameters=None*, *sld_profile=None*)

Create a single line diagram from a voltage level or a substation.

Parameters

- **container_id** (*str*) – a voltage level id or a substation id
- **parameters** (*SldParameters | None*) – single-line diagram parameters to adjust the rendering of the diagram
- **sld_profile** (*SldProfile | None*) – profile parameter to customize the single line diagram

Returns
the single line diagram

Return type
Svg

pypowsybl.network.Network.write_single_line_diagram_svg

Network.**write_single_line_diagram_svg**(*container_id*, *svg_file*, *metadata_file=None*, *parameters=None*, *sld_profile=None*)

Create a single line diagram in SVG format from a voltage level or a substation and write to a file.

Parameters

- **container_id** (*str*) – a voltage level id or a substation id
- **svg_file** (*str | PathLike*) – a svg file path
- **metadata_file** (*str | PathLike | None*) – a json metadata file path
- **parameters** (*SldParameters | None*) – single-line diagram parameters to adjust the rendering of the diagram
- **sld_profile** (*SldProfile | None*) – profile parameter to customize the single line diagram

Return type
None

pypowsybl.network.Network.get_matrix_multi_substation_single_line_diagram

Network.**get_matrix_multi_substation_single_line_diagram**(*matrix_ids*, *parameters=None*, *sld_profile=None*)

Create a single line diagram from multiple substations

Parameters

- **matrix_ids** (*List[List[str]]*) – a two-dimensional list of substation id
- **parameters** (*SldParameters | None*) – single-line diagram parameters to adjust the rendering of the diagram
- **sld_profile** (*SldProfile | None*) – profile parameter to customize the single line diagram

Returns
the single line diagram

Return type
Svg

pypowsybl.network.Network.write_matrix_multi_substation_single_line_diagram_svg

Network.**write_matrix_multi_substation_single_line_diagram_svg**(*matrix_ids*, *svg_file*, *metadata_file=None*, *parameters=None*, *sld_profile=None*)

Create a single line diagram in SVG format from a voltage level or a substation and write to a file.

Parameters

- **matrix_ids** (*List[List[str]]*) – a two-dimensional list of substation id
- **svg_file** (*str | PathLike*) – a svg file path
- **metadata_file** (*str | PathLike | None*) – a json metadata file path
- **parameters** (*SldParameters | None*) – single-line diagram parameters to adjust the rendering of the diagram
- **sld_profile** (*SldProfile | None*) – profile parameter to customize the single line diagram

Return type
None

pypowsybl.network.Network.get_network_area_diagram

Network.**get_network_area_diagram**(*voltage_level_ids=None*, *depth=0*, *high_nominal_voltage_bound=-1*, *low_nominal_voltage_bound=-1*, *nad_parameters=None*, *fixed_positions=None*, *nad_profile=None*)

Create a network area diagram.

Parameters

- **voltage_level_ids** (*List[str] | str | None*) – the voltage level IDs, centers of the diagram (None for the full diagram)
- **depth** (*int*) – the diagram depth around the voltage level
- **high_nominal_voltage_bound** (*float*) – high bound to filter voltage level according to nominal voltage
- **low_nominal_voltage_bound** (*float*) – low bound to filter voltage level according to nominal voltage
- **nad_parameters** (*NadParameters | None*) – parameters for network area diagram
- **fixed_positions** (*DataFrame | None*) – optional dataframe used to set fixed coordinates for diagram elements. Positions for elements not specified in the dataframe will be computed using the current layout.
- **nad_profile** (*NadProfile | None*) – parameters to customize the network area diagram

Returns
the network area diagram

Return type
Svg

pypowsybl.network.Network.write_network_area_diagram_svg

Network.**write_network_area_diagram_svg**(*svg_file*, *voltage_level_ids=None*, *depth=0*, *high_nominal_voltage_bound=-1*, *low_nominal_voltage_bound=-1*, *edge_name_displayed=False*)

Deprecated since version 1.1.0: Use *write_network_area_diagram* with *NadParameters* instead.

Create a network area diagram in SVG format and write it to a file. :param svg_file: a svg file path :param voltage_level_ids: the voltage level ID, center of the diagram (None for the full diagram) :param depth: the diagram depth around the voltage level :param high_nominal_voltage_bound: high bound to filter voltage level according to nominal voltage :param low_nominal_voltage_bound: low bound to filter voltage level according to nominal voltage :param edge_name_displayed: if true displays the edge's names

Parameters

- **svg_file** (*str | PathLike*)
- **voltage_level_ids** (*List[str] | str | None*)
- **depth** (*int*)
- **high_nominal_voltage_bound** (*float*)
- **low_nominal_voltage_bound** (*float*)
- **edge_name_displayed** (*bool*)

Return type
None

pypowsybl.network.Network.write_network_area_diagram

Network.**write_network_area_diagram**(*svg_file*, *voltage_level_ids=None*, *depth=0*, *high_nominal_voltage_bound=-1*, *low_nominal_voltage_bound=-1*, *nad_parameters=None*, *metadata_file=None*, *fixed_positions=None*, *nad_profile=None*)

Create a network area diagram in SVG format and write it to a file.

Parameters

- **svg_file** (*str | PathLike*) – a svg file path
- **metadata_file** (*str | PathLike | None*) – a json metadata file path (optional)
- **voltage_level_ids** (*List[str] | str | None*) – the voltage level ID, center of the diagram (None for the full diagram)
- **depth** (*int*) – the diagram depth around the voltage level
- **high_nominal_voltage_bound** (*float*) – high bound to filter voltage level according to nominal voltage
- **low_nominal_voltage_bound** (*float*) – low bound to filter voltage level according to nominal voltage
- **nad_parameters** (*NadParameters | None*) – parameters for network area diagram
- **fixed_positions** (*DataFrame | None*) – optional dataframe used to set fixed coordinates for diagram elements. Positions for elements not specified in the dataframe will be computed using the current layout.
- **nad_profile** (*NadProfile | None*) – parameters to customize the network area diagram

Return type
None

pypowsybl.network.Network.get_network_area_diagram_displayed_voltage_levels

Network.**get_network_area_diagram_displayed_voltage_levels**(*voltage_level_ids*, *depth=0*)

Gathers the name of the displayed voltage levels of a network-area diagram in a list, according to the input voltage level(s) and the depth of the diagram.

Parameters

- **voltage_level_ids** (*str | List[str]*) – the voltage level ID(s), center(s) of the diagram
- **depth** (*int*) – the diagram depth around the voltage level

Returns
a list of the displayed voltage levels

Return type
List[str]

pypowsybl.network.Network.get_default_nad_profile

Network.**get_default_nad_profile**()

Creates a default NadProfile for branch labels, twt labels, injection labels, bus descriptions, and VL descriptions.

Returns
a NadProfile where the labels and descriptions dataframes have been set with the content from the default NAD content provider.

Return type
NadProfile

pypowsybl.network.Network.disconnect

Network.**disconnect**(*id*, *operate_disconnectors=False*, *operate_fictitious=False*)

Disconnects a connectable element's terminal or terminals.

Parameters

- **id** (*str*) – The ID of the element to disconnect.
- **operate_disconnectors** (*bool*) – Whether disconnectors can be opened during disconnection.
- **operate_fictitious** (*bool*) – Whether fictitious switches can be opened during disconnection.

Returns
True if the disconnection was successful, False otherwise.

Return type
bool

pypowsybl.network.Network.connect

Network.**connect**(*id*, *operate_disconnectors=False*, *operate_fictitious=False*)

Connects a connectable element's terminal or terminals.

Parameters

- **id** (*str*) – The ID of the element to connect.
- **operate_disconnectors** (*bool*) – Whether disconnectors can be closed during connection.
- **operate_fictitious** (*bool*) – Whether fictitious switches can be closed during connection.

Returns
True if the connection was successful, False otherwise.

Return type
bool

pypowsybl.network.Network.open_switch

Network.**open_switch**(*id*)

Parameters
id (*str*)

Return type
bool

pypowsybl.network.Network.close_switch

Network.**close_switch**(*id*)

Parameters
id (*str*)

Return type
bool

pypowsybl.network.Network.get_validation_level

Network.**get_validation_level**()

The network's validation level.

This it the network validation level as computed by validation checks.

Returns
the ValidationLevel.

Return type
pypowsybl._pypowsybl.ValidationLevel

pypowsybl.network.Network.validate

Network.**validate**()

Validate the network.

The validation will raise an exception if any check is not consistent with the configured minimum validation level.

Returns
the computed ValidationLevel, which may be higher than the configured minimum level.

Raises
pypowsybl.PyPowsyblError – if any validation check is not consistent with the configured minimum validation level.

Return type
pypowsybl._pypowsybl.ValidationLevel

pypowsybl.network.Network.set_min_validation_level

Network.**set_min_validation_level**(*validation_level*)

Set the minimum validation level for the network.

Parameters
validation_level (*ValidationLevel*) – the validation level

Raises
pypowsybl.PyPowsyblError – if any validation check is not consistent with the new minimum validation level.

Return type
None

I/O

load	Load a network from a file.
load_from_string	Load a network from a string.
load_from_binary_buffer	Load a network from a binary buffer.
load_from_binary_buffers	Load a network from a list of binary buffers.
Network.update_from_file	Updates a network by loading information from a file.
Network.update_from_binary_buffer	Update a network from a binary buffer.
Network.update_from_binary_buffers	Update a network from a list of binary buffers.
Network.dump	
Network.dump_to_string	
get_import_formats	Get list of supported import formats
get_import_parameters	Supported import parameters for a given format.
get_import_post_processors	Get list of supported import post processors
get_export_formats	Get list of supported export formats
get_export_parameters	Get supported export parameters infos for a given format
Network.save	Save a network to a file using the specified format.
Network.save_to_string	Save a network to a string using a specified format.
Network.save_to_binary_buffer	Save a network to a binary buffer using a specified format.

pypowsybl.network.Network.update_from_file

Network.**update_from_file**(*file*, *parameters=None*, *post_processors=None*, *report_node=None*)

Updates a network by loading information from a file. File should be in a supported format.

Parameters

- **file** (*str | PathLike*) – path to the network file
- **parameters** (*Dict[str, str] | None*) – a dictionary of import parameters (optional)
- **post_processors** (*List[str] | None*) – a list of import post processors (optional, will be added to the ones defined by the platform config)
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Return type

None

pypowsybl.network.Network.update_from_binary_buffer

Network.**update_from_binary_buffer**(*buffer*, *parameters=None*, *post_processors=None*, *report_node=None*)

Update a network from a binary buffer.

Parameters

- **buffer** (*BytesIO*) – The BytesIO data buffer
- **parameters** (*Dict[str, str] | None*) – A dictionary of import parameters
- **post_processors** (*List[str] | None*) – a list of import post processors (will be added to the ones defined by the platform config)
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Return type

None

pypowsybl.network.Network.update_from_binary_buffers

Network.**update_from_binary_buffers**(*buffers*, *parameters=None*, *post_processors=None*, *report_node=None*)

Update a network from a list of binary buffers. Only zipped CGMES are supported for several zipped source load.

Parameters

- **buffers** (*List[BytesIO]*) – The BytesIO data buffers
- **parameters** (*Dict[str, str] | None*) – A dictionary of import parameters
- **post_processors** (*List[str] | None*) – a list of import post processors (will be added to the ones defined by the platform config)
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Return type

None

pypowsybl.network.Network.dump

Network.**dump**(*file*, *format='XIIDM'*, *parameters=None*, *reporter=None*)

Deprecated since version 1.1.0: Use *save()* instead.

Parameters

- **file** (*str | PathLike*)
- **format** (*str*)
- **parameters** (*Dict[str, str] | None*)
- **reporter** (*ReportNode | None*)

Return type
None

pypowsybl.network.Network.dump_to_string

Network.**dump_to_string**(*format='XIIDM'*, *parameters=None*, *reporter=None*)

Deprecated since version 1.1.0: Use *save_to_string()* instead.

Parameters

- **format** (*str*)
- **parameters** (*Dict[str, str] | None*)
- **reporter** (*ReportNode | None*)

Return type
str

pypowsybl.network.get_import_formats

get_import_formats()

Get list of supported import formats

Returns
the list of supported import formats

Return type
List[str]

pypowsybl.network.get_import_parameters

get_import_parameters(*fmt*)

Supported import parameters for a given format.

Parameters
fmt (*str*) – the format

Returns
import parameters dataframe

Return type
DataFrame

Examples

```
>>> parameters = pp.network.get_import_parameters('PSS/E')
>>> parameters.index.tolist()
['psse.import.ignore-base-voltage', 'psse.import.ignore-node-breaker-topology']
>>> parameters['description']['psse.import.ignore-base-voltage']
'Ignore base voltage specified in the file'
>>> parameters['type']['psse.import.ignore-base-voltage']
'BOOLEAN'
>>> parameters['default']['psse.import.ignore-base-voltage']
'false'
```

pypowsybl.network.get_import_post_processors

get_import_post_processors()

Get list of supported import post processors

Returns
the list of supported import post processors

Return type
List[str]

pypowsybl.network.get_export_formats

get_export_formats()

Get list of supported export formats

Returns
the list of supported export formats

Return type
List[str]

pypowsybl.network.get_export_parameters

get_export_parameters(*fmt*)

Get supported export parameters infos for a given format

Parameters
fmt (*str*) – the format

Returns
export parameters dataframe

Return type
DataFrame

pypowsybl.network.Network.save

Network.**save**(*file*, *format='XIIDM'*, *parameters=None*, *reporter=None*, *report_node=None*)

Save a network to a file using the specified format.

Basic compression formats are also supported: for example if file name ends with '.gz', the resulting files will be gzipped.

Parameters

- **file** (*str | PathLike*) – path to the exported file
- **format** (*str*) – format to save the network, defaults to 'XIIDM'
- **parameters** (*Dict[str, str] | None*) – a dictionary of export parameters
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Return type
None

Examples

Various usage examples:

```
network.save('network.xiidm')
network.save('network.xiidm.gz')  # produces a gzipped file
network.save('/path/to/network.uct', format='UCTE')
```

pypowsybl.network.Network.save_to_string

Network.**save_to_string**(*format='XIIDM'*, *parameters=None*, *reporter=None*, *report_node=None*)

Save a network to a string using a specified format.

Parameters

- **format** (*str*) – format to export, only support mono file type, defaults to 'XIIDM'
- **parameters** (*Dict[str, str] | None*) – a dictionary of export parameters
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Returns
A string representing this network

Return type
str

pypowsybl.network.Network.save_to_binary_buffer

Network.**save_to_binary_buffer**(*format='XIIDM'*, *parameters=None*, *reporter=None*, *report_node=None*)

Save a network to a binary buffer using a specified format. In the current implementation, whatever the specified format is (so a format creating a single file or a format creating multiple files), the created binary buffer is a zip file.

Parameters

- **format** (*str*) – format to export, only support mono file type, defaults to 'XIIDM'
- **parameters** (*Dict[str, str] | None*) – a dictionary of export parameters
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Returns
A BytesIO data buffer representing this network

Return type
BytesIO

Advanced network modifications

create_2_windings_transformer_bays	Creates a transformer and connects it to buses or busbar sections through standard feeder bays.
create_line_bays	Creates a line and connects it to buses or busbar sections through standard feeder bays.
create_load_bay	Creates a load, connects it to the network on a given bus or busbar section and creates the associated topology.
create_battery_bay	Creates a battery, connects it to the network on a given bus or busbar section and creates the associated topology.
create_boundary_line_bay	Creates a boundary line, connects it to the network on a given bus or busbar section and creates the associated topology.
create_dangling_line_bay	
create_generator_bay	Creates a generator, connects it to the network on a given bus or busbar section and creates the associated topology.
create_shunt_compensator_bay	Creates a shunt compensator, connects it to the network on a given bus or busbar section and creates the associated topology.
create_static_var_compensator_bay	Creates a static var compensator, connects it to the network on a given bus or busbar section and creates the associated topology.
create_lcc_converter_station_bay	Creates a lcc converter station, connects it to the network on a given bus or busbar section and creates the associated topology.
create_vsc_converter_station_bay	Creates a vsc converter station, connects it to the network on a given bus or busbar section and creates the associated topology.
create_line_on_line	Connects an existing voltage level to an existing line through a tee point.
revert_create_line_on_line	This method reverses the action done in the create_line_on_line method.
connect_voltage_level_on_line	Cuts an existing line in two lines and connects an existing voltage level between them.
revert_connect_voltage_level_on_line	This method reverses the action done in the connect_voltage_level_on_line method.
replace_tee_point_by_voltage_level_on_line	This method transforms the action done in the create_line_on_line function into the action done in the connect_voltage_level_on_line.
revert_connect_voltage_level_on_line	This method reverses the action done in the connect_voltage_level_on_line method.
create_voltage_level_topology	Creates the topology of a given symmetrical voltage level, containing a given number of busbar with a given number of sections.

continues on next page

Table 13 – continued from previous page

create_coupling_device	Creates a coupling device on the network between two busbar sections of a same voltage level.

pypowsybl.network.create_2_windings_transformer_bays

create_2_windings_transformer_bays(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a transformer and connects it to buses or busbar sections through standard feeder bays.

In node/breaker topology, the created bays are composed of one breaker, and one disconnector for each busbar section parallel to the section specified in arguments. Only the disconnector on the specified section is closed, others are left open.

In bus/breaker topology, the transformer is simply connected to the buses.

Parameters

- **network** (*Network*) – the network to which we want to add the new line
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

The input dataframe expects same attributes as *Network.create_2_windings_transformers()*, except for the additional following attributes:

- **bus_or_busbar_section_id_1**: the identifier of the bus or of the busbar section on side 1
- **position_order_1**: in node/breaker topology, the position of the transformer on side 1
- **direction_1**: optionally, in node/breaker, the direction, TOP or BOTTOM, of the transformer on side 1
- **bus_or_busbar_section_id_2**: the identifier of the bus or of the busbar section on side 2
- **position_order_2**: in node/breaker, the position of the transformer on side 2
- **direction_2**: optionally, in node/breaker, the direction, TOP or BOTTOM, of the transformer on side 2

Examples

```
pp.network.create_2_windings_transformers_bays(
                network, id='L', b=1e-6, g=1e-6, r=0.5, x=10, rated_u1=400, rated_
↪u2=225,
                bus_or_busbar_section_id_1='BBS1',
```

(continues on next page)

(continued from previous page)

```
                position_order_1=115,
                direction_1='TOP',
                bus_or_busbar_section_id_2='BBS2',
                position_order_2=121,
                direction_2='BOTTOM')
```

See also

Network.create_2_windings_transformers()

pypowsybl.network.create_line_bays

create_line_bays(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a line and connects it to buses or busbar sections through standard feeder bays.

In node/breaker topology, the created bays are composed of one breaker, and one disconnector for each busbar section parallel to the section specified in arguments. Only the disconnector on the specified section is closed, others are left open. In bus/breaker topology, the line is connected to the bus.

Parameters

- **network** (*Network*) – the network to which we want to add the new line
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

The input dataframe expects same attributes as *Network.create_lines()*, except for the additional following attributes:

- **bus_or_busbar_section_id_1**: the identifier of the bus or of the busbar section on side 1
- **position_order_1**: in node/breaker, the position of the line on side 1
- **direction_1**: optionally, in node/breaker, the direction, TOP or BOTTOM, of the line on side 1
- **bus_or_busbar_section_id_2**: the identifier of the bus or of the busbar section on side 2
- **position_order_2**: in node/breaker, the position of the line on side 2
- **direction_2**: optionally, in node/breaker, the direction, TOP or BOTTOM, of the line on side 2

Examples

```
pp.network.create_line_bays(network, id='L', r=0.1, x=10, g1=0, b1=0, g2=0, b2=0,
                            bus_or_busbar_section_id_1='BBS1',
                            position_order_1=115,
                            direction_1='TOP',
                            bus_or_busbar_section_id_2='BBS2',
                            position_order_2=121,
                            direction_2='BOTTOM')
```

See also

Network.create_lines()

pypowsybl.network.create_load_bay

create_load_bay(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a load, connects it to the network on a given bus or busbar section and creates the associated topology.

Parameters

- **network** (Network) – the network to which we want to add the load
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type

None

Notes

The voltage level containing the busbar section can be described in node/breaker or bus/breaker topology. If the voltage level is node/breaker, the load is connected to the busbar with a breaker and a closed disconnector. If the network has position extensions, the load will also be connected to every parallel busbar section with an open disconnector. If the voltage level is bus/breaker, the load is just connected to the bus.

Valid attributes are:

- **id**: the identifier of the new load
- **name**: an optional human-readable name
- **type**: optionally, the type of load (UNDEFINED, AUXILIARY, FICTITIOUS)
- **p0**: active power load, in MW
- **q0**: reactive power load, in MVar

- **bus_or_busbar_section_id**: id of the bus or of the busbar section to which the injection will be connected with a closed disconnector.
- **position_order**: in node/breaker, the order of the load, will fill the ConnectablePosition extension
- **direction**: optionally, in node/breaker, the direction of the load, will fill the ConnectablePosition extension, default is BOTTOM.

pypowsybl.network.create_battery_bay

`create_battery_bay`(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a battery, connects it to the network on a given bus or busbar section and creates the associated topology.

Parameters

- **network** (*Network*) – the network to which we want to add the battery
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type

None

Notes

The voltage level containing the busbar section can be described in node/breaker or bus/breaker topology. If the voltage level is node/breaker, the battery is connected to the busbar with a breaker and a closed disconnector. If the network has position extensions, the battery will also be connected to every parallel busbar section with an open disconnector. If the voltage level is bus/breaker, the battery is just connected to the bus.

Valid attributes are:

- **id**: the identifier of the new battery
- **name**: an optional human-readable name
- **min_p**: minimum active power, in MW
- **max_p**: maximum active power, in MW
- **target_p**: active power consumption, in MW
- **target_q**: reactive power consumption, in MVar
- **bus_or_busbar_section_id**: id of the bus or of the busbar section to which the injection will be connected with a closed disconnector.
- **position_order**: in node/breaker, the order of the battery, will fill the ConnectablePosition extension
- **direction**: optionally, in node/breaker, the direction of the battery, will fill the ConnectablePosition extension, default is BOTTOM.

pypowsybl.network.create_boundary_line_bay

create_boundary_line_bay(*network*, *df=None*, *generation_df=Empty DataFrame Columns: [] Index: []*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a boundary line, connects it to the network on a given bus or busbar section and creates the associated topology.

Parameters

- **network** (Network) – the network to which we want to add the boundary line
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **generation_df** (*DataFrame*) – Optional boundary lines' generation part, only as a dataframe
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Return type
None

Notes

The voltage level containing the busbar section can be described in node/breaker or bus/breaker topology. If the voltage level is node/breaker, the boundary line is connected to the busbar with a breaker and a closed disconnector. If the network has position extensions, the boundary line will also be connected to every parallel busbar section with an open disconnector. If the voltage level is bus/breaker, the boundary line is just connected to the bus.

Valid attributes for boundary line dataframe or named arguments are:

- **id**: the identifier of the new line
- **name**: an optional human-readable name
- **p0**: the active power consumption, in MW
- **q0**: the reactive power consumption, in MVar
- **r**: the resistance, in Ohms
- **x**: the reactance, in Ohms
- **g**: the shunt conductance, in S
- **b**: the shunt susceptance, in S
- **bus_or_busbar_section_id**: id of the bus or of the busbar section to which the injection will be connected with a closed disconnector.
- **position_order**: in node/breaker, the order of the boundary line, will fill the ConnectablePosition extension
- **direction**: optionally, in node/breaker, the direction of the boundary line, will fill the ConnectablePosition extension, default is BOTTOM.

Boundary line generation information must be provided as a dataframe. Valid attributes are:

- **id**: Identifier of the boundary line that contains this generation part
- **min_p**: Minimum active power output of the boundary line's generation part
- **max_p**: Maximum active power output of the boundary line's generation part
- **target_p**: Active power target of the generation part
- **target_q**: Reactive power target of the generation part
- **target_v**: Voltage target of the generation part
- **voltage_regulator_on**: `True` if the generation part regulates voltage

pypowsybl.network.create_dangling_line_bay

create_dangling_line_bay(*network*, *df=None*, *generation_df=Empty DataFrame Columns: [] Index: []*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Deprecated since version 1.15.0: Use *create_boundary_line_bay()* instead.

Parameters

- **network** (*Network*)
- **df** (*DataFrame | None*)
- **generation_df** (*DataFrame*)
- **raise_exception** (*bool*)
- **reporter** (*ReportNode | None*)
- **report_node** (*ReportNode | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

pypowsybl.network.create_generator_bay

create_generator_bay(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a generator, connects it to the network on a given bus or busbar section and creates the associated topology.

Parameters

- **network** (*Network*) – the network to which we want to add the generator
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).

- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

The voltage level containing the busbar section can be described in node/breaker or bus/breaker topology. If the voltage level is node/breaker, the generator is connected to the busbar with a breaker and a closed disconnector. If the network has position extensions, the generator will also be connected to every parallel busbar section with an open disconnector. If the voltage level is bus/breaker, the generator is just connected to the bus.

Valid attributes are:

- **id**: the identifier of the new generator
- **energy_source**: the type of energy source (HYDRO, NUCLEAR, ...)
- **max_p**: maximum active power in MW
- **min_p**: minimum active power in MW
- **target_p**: target active power in MW
- **target_q**: target reactive power in MVar, when the generator does not regulate voltage
- **rated_s**: nominal power in MVA
- **target_v**: target voltage in kV, when the generator regulates voltage
- **voltage_regulator_on**: true if the generator regulates voltage
- **bus_or_busbar_section_id**: id of the bus or of the busbar section to which the injection will be connected with a closed disconnector.
- **position_order**: in node/breaker, the order of the generator, will fill the ConnectablePosition extension
- **direction**: optionally, in node/breaker, the direction of the generator, will fill the ConnectablePosition extension, default is BOTTOM.

pypowsybl.network.create_shunt_compensator_bay

create_shunt_compensator_bay(*network*, *shunt_df*, *linear_model_df=None*, *non_linear_model_df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*)

Creates a shunt compensator, connects it to the network on a given bus or busbar section and creates the associated topology.

Parameters

- **network** (*Network*) – the network to which we want to add the shunt compensator
- **shunt_df** (*DataFrame*) – dataframe for shunt compensators data
- **linear_model_df** (*DataFrame | None*) – dataframe for linear model sections data
- **non_linear_model_df** (*DataFrame | None*) – dataframe for sections data
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead

- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).

Return type
None

Notes

The voltage level containing the busbar section can be described in node/breaker or bus/breaker topology. If the voltage level is node/breaker, the shunt compensator is connected to the busbar with a breaker and a closed disconnector. If the network has position extensions, the shunt compensator will also be connected to every parallel busbar section with an open disconnector. If the voltage level is bus/breaker, the shunt compensator is just connected to the bus.

Valid attributes for the shunt compensators dataframe are:

- **id**: the identifier of the new shunt
- **name**: an optional human-readable name
- **model_type**: either LINEAR or NON_LINEAR
- **section_count**: the current count of connected sections
- **target_v**: an optional target voltage in kV
- **target_v**: an optional deadband for the target voltage, in kV
- **bus_or_busbar_section_id**: id of the bus or of the busbar section to which the injection will be connected with a closed disconnector.
- **position_order**: in node/breaker, the order of the shunt compensator, will fill the ConnectablePosition extension
- **direction**: optionally, in node/breaker, the direction of the shunt compensator, will fill the ConnectablePosition extension, default is BOTTOM.

Valid attributes for the linear sections models are:

- **id**: the identifier of the new shunt
- **g_per_section**: the conductance, in Ohm, for each section
- **b_per_section**: the susceptance, in Ohm, for each section
- **max_section_count**: the maximum number of connectable sections

This dataframe must have only one row for each shunt compensator.

Valid attributes for the non-linear sections models are:

- **id**: the identifier of the new shunt
- **g**: the conductance, in Ohm, for this section
- **b**: the susceptance, in Ohm, for this section

pypowsybl.network.create_static_var_compensator_bay

create_static_var_compensator_bay(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a static var compensator, connects it to the network on a given bus or busbar section and creates the associated topology.

Parameters

- **network** (Network) – the network to which we want to add the static var compensator
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Return type
None

Notes

The voltage level containing the busbar section can be described in node/breaker or bus/breaker topology. If the voltage level is node/breaker, the static var compensator is connected to the busbar with a breaker and a closed disconnector. If the network has position extensions, the static var compensator will also be connected to every parallel busbar section with an open disconnector. If the voltage level is bus/breaker, the static var compensator is just connected to the bus.

Valid attributes are:

- **id**: the identifier of the new SVC
- **name**: an optional human-readable name
- **b_max**: the maximum susceptance, in S
- **b_min**: the minimum susceptance, in S
- **regulation_mode**: the regulation mode (VOLTAGE, REACTIVE_POWER, OFF)
- **target_v**: the target voltage, in kV, when the regulation mode is VOLTAGE
- **target_q**: the target reactive power, in MVar, when the regulation mode is not VOLTAGE
- **bus_or_busbar_section_id**: id of the bus or of the busbar section to which the injection will be connected with a closed disconnector.
- **position_order**: in node/breaker, the order of the static var compensator, will fill the ConnectablePosition extension
- **direction**: optionally, in node/breaker, the direction of the static var compensator, will fill the ConnectablePosition extension, default is BOTTOM.

pypowsybl.network.create_lcc_converter_station_bay

create_lcc_converter_station_bay(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a lcc converter station, connects it to the network on a given bus or busbar section and creates the associated topology.

Parameters

- **network** (Network) – the network to which we want to add the lcc converter station

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Return type
None

Notes

The voltage level containing the busbar section can be described in node/breaker or bus/breaker topology. If the voltage level is node/breaker, the lcc converter station is connected to the busbar with a breaker and a closed disconnector. If the network has position extensions, the lcc converter station will also be connected to every parallel busbar section with an open disconnector. If the voltage level is bus/breaker, the lcc converter station is just connected to the bus.

Valid attributes are:

- **id**: the identifier of the new station
- **name**: an optional human-readable name
- **power_factor**: the power factor (ratio of the active power to the apparent power)
- **loss_factor**: the loss factor of the station
- **bus_or_busbar_section_id**: id of the bus or of the busbar section to which the injection will be connected with a closed disconnector.
- **position_order**: in node/breaker, the order of the lcc converter station, will fill the ConnectablePosition extension
- **direction**: optionally, in node/breaker, the direction of the lcc converter station, will fill the ConnectablePosition extension, default is BOTTOM.

pypowsybl.network.create_vsc_converter_station_bay

create_vsc_converter_station_bay(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a vsc converter station, connects it to the network on a given bus or busbar section and creates the associated topology.

Parameters

- **network** (Network) – the network to which we want to add the vsc converter station
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead

- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – the data to be selected, as named arguments.

Return type
None

Notes

The voltage level containing the busbar section can be described in node/breaker or bus/breaker topology. If the voltage level is node/breaker, the vsc converter station is connected to the busbar with a breaker and a closed disconnector. If the network has position extensions, the vsc converter station will also be connected to every parallel busbar section with an open disconnector. If the voltage level is bus/breaker, the vsc converter station is just connected to the bus.

Valid attributes are:

- **id**: the identifier of the new station
- **name**: an optional human-readable name
- **loss_factor**: the loss factor of the new station
- **voltage_regulator_on**: true if the station regulated voltage
- **target_v**: the target voltage, in kV, when the station regulates voltage
- **target_q**: the target reactive power, in MVar, when the station does not regulate voltage
- **bus_or_busbar_section_id**: id of the bus or of the busbar section to which the injection will be connected with a closed disconnector.
- **position_order**: in node/breaker, the order of the vsc converter station, will fill the ConnectablePosition extension
- **direction**: optionally, in node/breaker, the direction of the vsc converter station, will fill the ConnectablePosition extension, default is BOTTOM.

pypowsybl.network.create_line_on_line

create_line_on_line(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Connects an existing voltage level to an existing line through a tee point.

Connects an existing voltage level (in practice a voltage level where we have some loads or generations) to an existing line through a tee point. This method cuts an existing line in two, creating a fictitious voltage level between them (on a fictitious substation if asked). Then it links an existing voltage level to this fictitious voltage level by creating a new line described in the provided dataframe.

Parameters

- **network** (Network) – the network
- **df** (*DataFrame | None*) – attributes as a dataframe, it should contain:
  - bbs_or_bus_id: the ID of the existing bus or bus bar section of the voltage level voltage_level_id.
  - new_line_id: ID of the new line
  - new_line_r: resistance of the new line, in ohms

- new_line_x: reactance of the new line, in ohms
- new_line_b1: shunt susceptance on side 1 of the new line
- new_line_b2: shunt susceptance on side 2 of the new line
- new_line_g1: shunt conductance on side 1 of the new line
- new_line_g2: shunt conductance on side 2 of the new line
- line_id: the id on of the line on which we want to create a tee point.
- line1_id: when the initial line is cut, the line segment at side 1 has a given ID (optional).
- line1_name: when the initial line is cut, the line segment at side 1 has a given name (optional).
- line2_id: when the initial line is cut, the line segment at side 2 has a given ID (optional).
- line2_name: when the initial line is cut, the line segment at side 2 has a given name (optional).
- position_percent: when the existing line is cut in two lines, percent is equal to the ratio between the parameters of the first line and the parameters of the line that is cut multiplied by 100. 100 minus percent is equal to the ratio between the parameters of the second line and the parameters of the line that is cut multiplied by 100.
- create_fictitious_substation: True to create the fictitious voltage level inside a fictitious substation (false by default).
- fictitious_voltage_level_id: the ID of the fictitious voltage level (optional) containing the tee point.
- fictitious_voltage_level_name: the name of the fictitious voltage level (optional) containing the tee point.
- fictitious_substation_id: the ID of the fictitious substation (optional).
- fictitious_substation_name: the name of the fictitious substation (optional).

- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report). bbs_or_bus_id: the ID of the existing bus or bus bar section of the voltage level voltage_level_id.
- ****kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – attributes as keyword arguments

Return type
None

pypowsybl.network.revert_create_line_on_line

revert_create_line_on_line(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

This method reverses the action done in the create_line_on_line method. It replaces 3 existing lines (with the same voltage level as the one on their side) with a new line, and eventually removes the existing voltage levels (tee point and tapped voltage level), if they contain no equipments anymore, except bus or bus bar section.

Parameters

- **network** (Network) – the network
- **df** (*DataFrame | None*) – attributes as a dataframe, it should contain:
  - line_to_be_merged1_id: The id of the first line connected to the tee point.
  - line_to_be_merged2_id: The id of the second line connected to the tee point.
  - line_to_be_deleted: The tee point line that will be deleted
  - merged_line_id: The id of the new line from the two lines to be merged
  - merged_line_name: The name of the new line from the two lines to be merged (default to line id)
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- ****kwargs** (*str*) – attributes as keyword arguments
- **report_node** (*ReportNode | None*)
- ****kwargs**

Return type
None

pypowsybl.network.connect_voltage_level_on_line

connect_voltage_level_on_line(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Cuts an existing line in two lines and connects an existing voltage level between them.

This method cuts an existing line in two lines and connect an existing voltage level between them. The voltage level should be added to the network just before calling this method, and should contain at least a configured bus in bus/breaker topology or a bus bar section in node/breaker topology.

Parameters

- **network** (Network) – the network
- **df** (*DataFrame | None*) – attributes as a dataframe, it should contain:
  - bbs_or_bus_id: The ID of the configured bus or bus bar section to which the lines will be connected.
  - line_id: the ID ot the line on which the voltage level should be connected.
  - **position_percent: when the existing line is cut, percent is equal to the ratio between the parameters of the first line**
    and the parameters of the line that is cut multiplied by 100. 100 minus percent is equal to the ratio between the parameters of the second line and the parameters of the line that is cut multiplied by 100.
  - line1_id: when the initial line is cut, the line segment at side 1 will receive this ID (optional).
  - line1_name: when the initial line is cut, the line segment at side 1 will receive this name (optional).

  - line2_id: when the initial line is cut, the line segment at side 2 will receive this ID (optional).
  - line2_name: when the initial line is cut, the line segment at side 2 will receive this name (optional).
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- ****kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – attributes as keyword arguments
- **report_node** (*ReportNode | None*)
- ****kwargs**

Return type
None

pypowsybl.network.revert_connect_voltage_level_on_line

revert_connect_voltage_level_on_line(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

This method reverses the action done in the connect_voltage_level_on_line method. It replaces 2 existing lines (with the same voltage level at one of their side) with a new line, and eventually removes the voltage level in common (switching voltage level), if it contains no equipments anymore, except bus or bus bar section.

Parameters

- **network** (Network) – the network
- **df** (*DataFrame | None*) – attributes as a dataframe, it should contain: line1_id: The id of the first existing line line2_id: The id of the second existing line line_id: The id of the new line to be created line_name: The name of the line to be created (default to line_id)
- **raise_exception** (*bool*) – optionally, whether the calculation should throw exceptions. In any case, errors will be logged. Default is True.
- **reporter** (*ReportNode | None*) – optionally, the reporter to be used to create an execution report, default is None (no report).
- ****kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – attributes as keyword arguments
- **report_node** (*ReportNode | None*)
- ****kwargs**

Return type
None

pypowsybl.network.replace_tee_point_by_voltage_level_on_line

replace_tee_point_by_voltage_level_on_line(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

This method transforms the action done in the create_line_on_line function into the action done in the connect_voltage_level_on_line.

Parameters

- **network** (Network) – the network in which the busbar sections are.
- **df** (*DataFrame | None*) – Attributes as a dataframe. It should contain:
  - tee_point_line1: The ID of the existing line connecting the first voltage level to the tee point
  - tee_point_line2: The ID of the existing line connecting the tee point to the second voltage level
  - tee_point_line_to_remove: The ID of the existing line connecting the tee point to the attached voltage level
  - bbs_or_bus_id: The ID of the existing bus or bus bar section in the attached voltage level voltageLevelId, where we want to connect the new lines new line 1 and new line 2
  - new_line1_id: The ID of the new line connecting the first voltage level to the attached voltage level
  - new_line2_id: The ID of the new line connecting the second voltage level to the attached voltage level
  - new_line1_name: The optional name of the new line connecting the first voltage level to the attached voltage level
  - new_line2_name: The optional name of the new line connecting the second voltage level to the attached voltage level
- **raise_exception** (*bool*) – whether an exception should be raised if a problem occurs. By default, true.
- **reporter** (*ReportNode | None*) – an optional reporter to get functional logs.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – attributes as keyword arguments.
- **report_node** (*ReportNode | None*)

Return type
None

Notes

It replaces 3 existing lines (with the same voltage level at one of their side (tee point)) with two new lines, and removes the tee point.

pypowsybl.network.create_voltage_level_topology

create_voltage_level_topology(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates the topology of a given symmetrical voltage level, containing a given number of busbar with a given number of sections.

Parameters

- **network** (Network) – the network in which the busbar sections are.

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – whether an exception should be raised if a problem occurs. By default, true.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – an optional reporter to get functional logs.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – attributes as keyword arguments.

Return type
None

Notes

The voltage level must be created and in node/breaker or bus/breaker topology. In node/breaker topology, busbar sections will be created, as well as disconnectors or breakers between each section depending on the switch_kind list. In bus/breaker topology, a matrix of buses will be created containing section_count x aligned_buses_or_busbar_count buses. The buses on the same row of the matrix will be connected via a breaker.

The input dataframe expects these attributes: - **voltage_level_id**: the identifier of the voltage level where the topology should be created. - **low_bus_or_busbar_index**: the lowest bus or busbar index to be used. By default, 1 (no other buses or busbar sections). - **aligned_buses_or_busbar_count**: the total number of busbar or rows of buses to be created. - **low_section_index**: the lowest section index to be used. By default, 1. - **bus_or_busbar_section_prefix_id**: an optional prefix to put on the names of the created buses or busbar sections. By default, nothing. - **switch_prefix_id**: an optional prefix to put on the names of the created switches. By default, nothing. - **switch_kinds**: string or list containing the type of switch between each section. It should contain section_count - 1 switches and should look like that 'BREAKER, DISCONNECTOR' or ['BREAKER', 'DISCONNECTOR']. - **section_count**: optionally in node/breaker, required in bus/breaker, the number of sections to be created.

Examples:

```
pp.network.create_voltage_level_topology(network=network, raise_exception=True, id=
↪'VL',
                                         aligned_buses_or_busbar_count=3, switch_
↪kinds='BREAKER, DISCONNECTOR')
```

pypowsybl.network.create_coupling_device

create_coupling_device(*network*, *df=None*, *raise_exception=True*, *reporter=None*, *report_node=None*, ***kwargs*)

Creates a coupling device on the network between two busbar sections of a same voltage level.

Parameters

- **network** (Network) – the network in which the busbar sections are.
- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **raise_exception** (*bool*) – an optional boolean indicating if an exception should be raised in case an error occurs during
- **default** (*computation. By*)
- **true.**
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead

- **report_node** (*ReportNode | None*) – an optional reporter to store the funtional logs.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

The voltage level containing the busbar sections can be described in node/breaker or bus/breaker topology. In node/breaker topology, a closed breaker will be created as well as a closed disconnector on both given busbar sections to connect them. If the topology extensions are present on the busbar sections then on every parallel busbar section, an open disconnectors will be created to connect them to the breaker. If the two given busbar sections are the only two parallel busbar sections, and they have the same section index then, only two closed disconnectors will be created. In bus/breaker topology, a closed breaker will be created between two buses.

The input dataframe expects these attributes:

- **bus_or_busbar_section_id_1**: the identifier of the bus or of the busbar section on side 1
- **bus_or_busbar_section_id_2**: the identifier of the bus or of the busbar section on side 2
- **switch_prefix_id**: an optional prefix for all the switches

Examples

```
pp.network.create_coupling_device(
                network, bus_or_busbar_section_id_1='BBS1', bus_or_busbar_section_
↪id_2='BBS2',
                switch_prefix_id='sw')
```

Utility functions

get_connectables_order_positions	Gets the order positions of every connectable of a given voltage level in a dataframe.
get_unused_order_positions_before	Gets all the available order positions before a busbar section.
get_unused_order_positions_after	Gets all the available order positions after a busbar section.

pypowsybl.network.get_connectables_order_positions

get_connectables_order_positions(*network*, *voltage_level_id*)

Gets the order positions of every connectable of a given voltage level in a dataframe.

Parameters

- **network** (Network) – the network containing the voltage level.
- **voltage_level_id** (*str*) – id of the voltage level for which we want to get the order positions.

Return type
DataFrame

> **Note**
>
> About order positions: order positions represent the relative positions of every connectable compared to each other on a busbar section. It is filled in the ConnectablePosition extension under Order. Each connectable has as many positions as it has feeders. In this method, we get all the taken order positions by every connectable at the scale of a voltage level.

pypowsybl.network.get_unused_order_positions_before

`get_unused_order_positions_before`(*network*, *busbar_section_id*)

Gets all the available order positions before a busbar section.

Parameters

- **network** (Network) – the network in which the busbar section is.
- **busbar_section_id** (*str*) – the id of the busbar section from which we want to get all the available positions before.

Return type

Interval | None

Notes

Gets all the available positions between the lowest used position of a given busbar section and the highest used position of the busbar section with the section index equal to the section index of the given busbar section minus one. The result is an interval that includes the lowest available value and the highest available value. About order positions: order positions represent the relative positions of every connectable compared to each other on a busbar section. It is filled in the ConnectablePosition extension under Order. Each connectable has as many positions as it has feeders.

Examples

Let's take two busbar sections. The first one, bbs1, has 3 feeders with taken order positions 5,6,7. The second, bbs2 has two feeder with taken order positions 11 and 12. Then, get_unused_order_positions_before(bbs1) will return [-infinity, 4] as an interval and get_unused_order_positions_before(bbs2) will return [8,10] as an interval.

pypowsybl.network.get_unused_order_positions_after

`get_unused_order_positions_after`(*network*, *busbar_section_id*)

Gets all the available order positions after a busbar section.

Parameters

- **network** (Network) – the network in which the busbar section is.
- **busbar_section_id** (*str*) – the id of the busbar section from which we want to get all the available positions after.

Return type

Interval | None

Notes

Gets all the available positions between the highest used position of a given busbar section and the lowest used position of the busbar section with the section index equal to the section index of the given busbar section plus one. The result is a list with the lowest available value as the first integer and the highest available value as

the second integer. About order positions: order positions represent the relative positions of every connectable compared to each other on a busbar section. It is filled in the ConnectablePosition extension under Order. Each connectable has as many positions as it has feeders.

Examples

Let's take two busbar sections. The first one, bbs1, has 3 feeders with taken order positions 5,6,7. The second, bbs2 has two feeder with taken order positions 11 and 12. Then, get_unused_order_positions_after(bbs1) will return [8, 10] as an interval and get_unused_order_positions_before(bbs2) will return [13, +infinity] as an interval.

Scalable network modifications

These network modifications can be created to apply active power change across one or multiple injections

Scalable.scale	Apply the active power scaling to a given network.
InjectionScalable	Scalable based on a single injection.
StackScalable	Scalable that applies the asked modification on a list of underlying scalables, using them in order.
StackScalable.from_injections	Create a StackScalable from a list of injection ids.
StackScalable.from_scalables	Create a StackScalable from a list of underlying Scalable.
ProportionalScalable	Scalable based on a proportional repartition of power change between underlying elements.
ProportionalScalable.from_injections_and_percentages	Create a ProportionalScalable from a list of injection IDs and their corresponding percentage of power repartition.
ProportionalScalable.from_scalables_and_percentages	Create a ProportionalScalable from a list of Scalable and their corresponding percentage of power repartition.
ProportionalScalable.from_injections_and_distribution_mode	Create a ProportionalScalable from a list of injection IDs, computing their respective percentage of power repartition according to a given distribution mode.
UpDownScalable	Scalable based on two others : one used when scaling the power up and one to scale the power down
UpDownScalable.from_injections	Create an UpDownScalable from two injection ids.
UpDownScalable.from_scalables	Create an UpDownScalable from two Scalable.

pypowsybl.network.scalable.Scalable.scale

Scalable.**scale**(*network*, *asked*, *parameters=ScalingParameters(scaling_convention=pypowsybl._pypowsybl.ScalingParameters.scaling_convention, constant_power_factor=pypowsybl._pypowsybl.ScalingParameters.constant_power_factor, reconnect=pypowsybl._pypowsybl.ScalingParameters.reconnect, allows_generator_out_of_active_power_limits=pypowsybl._pypowsybl.ScalingParameters.allows_generator_out_of_a priority=pypowsybl._pypowsybl.ScalingParameters.priority, scaling_type=pypowsybl._pypowsybl.ScalingParameters.scaling_type, ignored_injection_ids=pypowsybl._pypowsybl.ScalingParameters.ignored_injection_ids)*)

Apply the active power scaling to a given network.

Parameters

- **network** (Network) – The network on which to apply the scaling
- **asked** (*float*) – The asked scaling value (in MW, delta or target depending on parameters)
- **parameters** (ScalingParameters) – Scaling parameters

Returns
The actual active power value applied (still in delta or target depending on parameters)

Return type
float

pypowsybl.network.scalable.InjectionScalable

class InjectionScalable(*injection_id*, *min_value=None*, *max_value=None*)

Scalable based on a single injection.

Parameters

- **injection_id** (*str*) – The id of the injection the scalable will modify
- **min_value** (*optional*) – The minimum active power value the modification can reach
- **max_value** (*optional*) – The maximum active power value the modification can reach

Methods

__init__(injection_id[, min_value, max_value])	
scale(network, asked[, parameters])	Apply the active power scaling to a given network.

Attributes

injection_id
max_value
min_value
type

pypowsybl.network.scalable.StackScalable

class StackScalable(*scalables*, *min_value=None*, *max_value=None*)

Scalable that applies the asked modification on a list of underlying scalables, using them in order. The power change is applied to the first Scalable in the list until it is at its limit, then to the next one.

Methods

__init__(scalables[, min_value, max_value])	
from_injections(injection_ids[, min_value, ...])	Create a StackScalable from a list of injection ids.
from_scalables(scalables[, min_value, max_value])	Create a StackScalable from a list of underlying Scalable.
scale(network, asked[, parameters])	Apply the active power scaling to a given network.

Attributes

children
max_value
min_value

continues on next page

Table 19 – continued from previous page

type

Parameters

- **scalables** (*List[Scalable]*)
- **min_value** (*float*)
- **max_value** (*float*)

pypowsybl.network.scalable.StackScalable.from_injections

classmethod StackScalable.**from_injections**(*injection_ids*, *min_value=None*, *max_value=None*)

Create a StackScalable from a list of injection ids.

Parameters

- **injection_ids** (*List[str]*) – The list of injection ids with which to create the underlying Scalable
- **min_value** (*optional*) – The minimum active power value the modification can reach
- **max_value** (*optional*) – The maximum active power value the modification can reach

Return type
StackScalable

pypowsybl.network.scalable.StackScalable.from_scalables

classmethod StackScalable.**from_scalables**(*scalables*, *min_value=None*, *max_value=None*)

Create a StackScalable from a list of underlying Scalable.

Parameters

- **scalables** (*List[Scalable]*) – The list of underlying Scalable to stack
- **min_value** (*optional*) – The minimum active power value the modification can reach
- **max_value** (*optional*) – The maximum active power value the modification can reach

Return type
StackScalable

pypowsybl.network.scalable.ProportionalScalable

class ProportionalScalable(*scalables*, *percentages*, *min_value=None*, *max_value=None*)

Scalable based on a proportional repartition of power change between underlying elements.

Methods

__init__(scalables, percentages[, ...])	
compute_scalables_percentages(injection_ids, ...)	
from_injections_and_distribution_mode(...[, ...])	Create a ProportionalScalable from a list of injection IDs, computing their respective percentage of power repartition according to a given distribution mode.

continues on next page

Table 20 – continued from previous page

from_injections_and_percentages(...[, ...])	Create a ProportionalScalable from a list of injection IDs and their corresponding percentage of power repartition.
from_scalables_and_percentages(scalables, ...)	Create a ProportionalScalable from a list of Scalable and their corresponding percentage of power repartition.
scale(network, asked[, parameters])	Apply the active power scaling to a given network.

Attributes

children
max_value
min_value
percentages
type

Parameters

- **scalables** (*List[Scalable]*)
- **percentages** (*List[float]*)
- **min_value** (*float*)
- **max_value** (*float*)

pypowsybl.network.scalable.ProportionalScalable.from_injections_and_percentages

classmethod ProportionalScalable.**from_injections_and_percentages**(*injection_ids*, *percentages*, *min_value=None*, *max_value=None*)

Create a ProportionalScalable from a list of injection IDs and their corresponding percentage of power repartition.

Parameters

- **injection_ids** (*List[str]*) – List of injection IDs.
- **percentages** (*List[float]*) – List of percentages corresponding to each injection.
- **min_value** (*float | None*) – Minimum value for the scalable. Defaults to None.
- **max_value** (*float | None*) – Maximum value for the scalable. Defaults to None.

Return type
ProportionalScalable

pypowsybl.network.scalable.ProportionalScalable.from_scalables_and_percentages

classmethod ProportionalScalable.**from_scalables_and_percentages**(*scalables*, *percentages*, *min_value=None*, *max_value=None*)

Create a ProportionalScalable from a list of Scalable and their corresponding percentage of power repartition.

Parameters

- **scalables** (*List[Scalable]*) – List of underlying Scalable.

- **percentages** (*List[float]*) – List of percentages corresponding to each injection.
- **min_value** (*float | None*) – Minimum value for the scalable. Defaults to None.
- **max_value** (*float | None*) – Maximum value for the scalable. Defaults to None.

Return type
ProportionalScalable

pypowsybl.network.scalable.ProportionalScalable.from_injections_and_distribution_mode

classmethod ProportionalScalable.**from_injections_and_distribution_mode**(*injection_ids*, *network*, *mode*, *min_value=None*, *max_value=None*)

Create a ProportionalScalable from a list of injection IDs, computing their respective percentage of power repartition according to a given distribution mode.

For example, with UNIFORM_DISTRIBUTION, all injections will have the same percentage, which is 1 / len(injection_ids).

Parameters

- **injection_ids** (*List[str]*) – List of injection IDs.
- **network** (Network) – Network with which to compute the percentages according to "mode".
- **mode** (*pypowsybl._pypowsybl.DistributionMode*) – Distribution mode to use for computing percentages The available modes are : UNIFORM_DISTRIBUTION, PROPORTIONAL_TO_TARGETP, PROPORTIONAL_TO_PMAX, PROPORTIONAL_TO_DIFF_PMAX_TARGETP, PROPORTIONAL_TO_DIFF_TARGETP_PMIN, PROPORTIONAL_TO_P0
- **min_value** (*float | None*) – Minimum value for the scalable. Defaults to None.
- **max_value** (*float | None*) – Maximum value for the scalable. Defaults to None.

Return type
ProportionalScalable

pypowsybl.network.scalable.UpDownScalable

class UpDownScalable(*up_scalable*, *down_scalable*, *min_value=None*, *max_value=None*)

Scalable based on two others : one used when scaling the power up and one to scale the power down

Methods

__init__(up_scalable, down_scalable[, ...])	
from_injections(up_injection_id, ...[, ...])	Create an UpDownScalable from two injection ids.
from_scalables(up_scalable, down_scalable[, ...])	Create an UpDownScalable from two Scalable.
scale(network, asked[, parameters])	Apply the active power scaling to a given network.

Attributes

max_value
min_value

continues on next page

Table 23 – continued from previous page

up_scalable
down_scalable
type

Parameters

- **up_scalable** (*Scalable*)
- **down_scalable** (*Scalable*)
- **min_value** (*float*)
- **max_value** (*float*)

pypowsybl.network.scalable.UpDownScalable.from_injections

classmethod UpDownScalable.**from_injections**(*up_injection_id*, *down_injection_id*, *min_value=None*, *max_value=None*)

Create an UpDownScalable from two injection ids.

Parameters

- **up_injection_id** (*str*) – The id of the injection with which to create the up scalable
- **down_injection_id** (*str*) – The id of the injection with which to create the down scalable
- **min_value** (*optional*) – The minimum active power value the modification can reach
- **max_value** (*optional*) – The maximum active power value the modification can reach

Return type
UpDownScalable

pypowsybl.network.scalable.UpDownScalable.from_scalables

classmethod UpDownScalable.**from_scalables**(*up_scalable*, *down_scalable*, *min_value=None*, *max_value=None*)

Create an UpDownScalable from two Scalable.

Parameters

- **up_scalable** (*Scalable*) – The Scalable used to up power
- **down_scalable** (*Scalable*) – The Scalable used to lower power
- **min_value** (*optional*) – The minimum active power value the modification can reach
- **max_value** (*optional*) – The maximum active power value the modification can reach

Return type
UpDownScalable

The `scale` method is parametrized by a ScalingParameters object :

ScalingParameters	Parameters to modify active power with a Scalable.

pypowsybl.network.scalable.ScalingParameters

class ScalingParameters(*scaling_convention=None*, *constant_power_factor=None*, *reconnect=None*, *allows_generator_out_of_active_power_limits=None*, *priority=None*, *scaling_type=None*, *ignored_injection_ids=None*)

Parameters to modify active power with a Scalable.

Parameters

- **scaling_convention** (*pypowsybl._pypowsybl.ScalingConvention | None*) – The scaling convention to use. The default is GENERATOR_CONVENTION and can be changed to LOAD_CONVENTION.
- **constant_power_factor** (*bool | None*) – If True, the scaling is done with a constant power factor (False by default)
- **reconnect** (*bool | None*) – If True, scaling can reconnect the underlying injection if it is disconnected (False by default)
- **allows_generator_out_of_active_power_limits** (*bool | None*) – If True, the scalable can modify the active power of a generator even if it is out of its active power limits (False by default)
- **priority** (*pypowsybl._pypowsybl.Priority | None*) – representing the priority of the scaling for ProportionalScalable. It can be either :
  - RESPECT_OF_VOLUME_ASKED (the scaling will distribute the power asked as much as possible by iterating if elements get saturated, even if it means not respecting potential percentages)
  - RESPECT_OF_DISTRIBUTION (the scaling will respect the percentages even if it means not scaling all what is asked)
  - ONESHOT (the scaling will distribute the power asked as is, in one iteration even if elements get saturated and even if it means not respecting potential percentages).

  (ONESHOT by default)
- **scaling_type** (*pypowsybl._pypowsybl.ScalingType | None*) – The type of scaling to use. The default is DELTA_P and can be changed to TARGET_P.
- **ignored_injection_ids** (*List[str] | None*) – List of injection ids to ignore when scaling.

Methods

__init__([scaling_convention, ...])	

3.1.2 Loadflow

The loadflow module allows to run AC and DC loadflows. It also provides a method to check the consistency of a network with loadflow equations.

Running a loadflow

run_ac	Run an AC load flow on a network.
run_dc	Run a DC load flow on a network.

continues on next page

Table 26 – continued from previous page

set_default_provider	Set the default load flow provider
get_default_provider	Get the current default loadflow provider.
get_provider_names	Get list of supported provider names.
get_provider_parameters_names	Get list of parameters for the specified loadflow provider.
get_provider_parameters	Supported loadflow specific parameters for a given provider.

pypowsybl.loadflow.run_ac

run_ac(*network*, *parameters=None*, *provider=''*, *reporter=None*, *report_node=None*)

Run an AC load flow on a network.

Parameters

- **network** (Network) – a network
- **parameters** (Parameters | *None*) – the load flow parameters, dc attribute is forced to false
- **provider** (*str*) – the load flow implementation provider, default is the default load flow provider
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Returns

A list of component results, one for each component of the network.

Return type

List[ComponentResult]

pypowsybl.loadflow.run_dc

run_dc(*network*, *parameters=None*, *provider=''*, *reporter=None*, *report_node=None*)

Run a DC load flow on a network.

Parameters

- **network** (Network) – a network
- **parameters** (Parameters | *None*) – the load flow parameters, dc attribute is forced to true
- **provider** (*str*) – the load flow implementation provider, default is the default load flow provider
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Returns

A list of component results, one for each component of the network.

Return type

List[ComponentResult]

pypowsybl.loadflow.set_default_provider

set_default_provider(*provider*)

Set the default load flow provider

Parameters
provider (*str*) – name of the default load flow provider to set

Return type
None

pypowsybl.loadflow.get_default_provider

get_default_provider()

Get the current default loadflow provider. if nothing is set it is OpenLoadFlow

Returns
the name of the current default loadflow provider

Return type
str

pypowsybl.loadflow.get_provider_names

get_provider_names()

Get list of supported provider names.

Returns
the list of supported provider names

Return type
List[str]

pypowsybl.loadflow.get_provider_parameters_names

get_provider_parameters_names(*provider=None*)

Get list of parameters for the specified loadflow provider.

Parameters
provider (*str*) – the provider, if not specified the provider will be the default one.

Returns
the list of provider's parameters

Return type
List[str]

pypowsybl.loadflow.get_provider_parameters

get_provider_parameters(*provider=None*)

Supported loadflow specific parameters for a given provider.

Parameters
provider (*str*) – the provider, if not specified the provider will be the default one.

Returns
loadflow parameters dataframe

Return type
DataFrame

Examples

```
>>> parameters = pp.loadflow.get_provider_parameters('OpenLoadFlow')
>>> parameters['category_key']['slackBusSelectionMode']
'SlackDistribution'
>>> parameters['description']['slackBusSelectionMode']
'Slack bus selection mode'
>>> parameters['type']['slackBusSelectionMode']
'STRING'
>>> parameters['default']['slackBusSelectionMode']
'MOST_MESHED'
>>> parameters['possible_values']['slackBusSelectionMode']
'[FIRST, MOST_MESHED, NAME, LARGEST_GENERATOR]'
```

Parameters

The execution of the loadflow can be customized using loadflow parameters.

Parameters	Parameters for a loadflow execution.
get_provider_parameters_names	Get list of parameters for the specified loadflow provider.
get_provider_parameters	Supported loadflow specific parameters for a given provider.
Parameters.to_json	Creates JSON string representation of the Parameters.
Parameters.from_json	Creates a Parameters object from a JSON string.

pypowsybl.loadflow.Parameters

class Parameters(*voltage_init_mode=None*, *transformer_voltage_control_on=None*, *use_reactive_limits=None*, *phase_shifter_regulation_on=None*, *twt_split_shunt_admittance=None*, *shunt_compensator_voltage_control_on=None*, *read_slack_bus=None*, *write_slack_bus=None*, *distributed_slack=None*, *balance_type=None*, *dc_use_transformer_ratio=None*, *countries_to_balance=None*, *component_mode=None*, *connected_component_mode=None*, *dc_power_factor=None*, *hvdc_ac_emulation=None*, *dc=None*, *provider_parameters=None*)

Parameters for a loadflow execution.

All parameters are first read from your configuration file, then overridden with the constructor arguments.

Please note that loadflow providers may not honor all parameters, according to their capabilities. For example, some providers will not be able to simulate the voltage control of shunt compensators, etc. The exact behaviour of some parameters may also depend on your loadflow provider. Please check the documentation of your provider for that information.

Parameters

- **voltage_init_mode** (*pypowsybl._pypowsybl.VoltageInitMode | None*) – The resolution starting point. Use `UNIFORM_VALUES` for a flat start, and `DC_VALUES` for a DC load flow based starting point.
- **transformer_voltage_control_on** (*bool | None*) – Simulate transformer voltage control. The initial tap position is used as starting point for the resolution.

- **use_reactive_limits** (*bool | None*) – Use reactive limits (named no_generator_reactive_limits with inverted logic before PyPowSyBl 1.3.0).
- **phase_shifter_regulation_on** (*bool | None*) – Simulate phase shifters regulation.
- **twt_split_shunt_admittance** (*bool | None*) – Split shunt admittance of transformers on both sides. Change the modelling of transformer legs. If you want to split the conductance and the susceptance in two, one at each side of the serie impedance, use `True`.
- **shunt_compensator_voltage_control_on** (*bool | None*) – Simulate voltage control of shunt compensators (named simul_shunt before PyPowSyBl 1.3.0).
- **read_slack_bus** (*bool | None*) – Read slack bus from the network. The slack bus needs to be defined through a dedicate extension. Prefer `False` if you want to use your loadflow provider selection mechanism, typically the most meshed bus.
- **write_slack_bus** (*bool | None*) – Write selected slack bus to the network. Will tag the slack bus selected by your loadflow provider with an extension.
- **distributed_slack** (*bool | None*) – Distribute active power slack on the network. `True` means that the active power slack is distributed, on loads or on generators according to `balance_type`.
- **balance_type** (*pypowsybl._pypowsybl.BalanceType | None*) – How to distribute active power slack. Use `PROPORTIONAL_TO_LOAD` to distribute slack on loads, `PROPORTIONAL_TO_GENERATION_P_MAX` or `PROPORTIONAL_TO_GENERATION_P` to distribute on generators.
- **dc_use_transformer_ratio** (*bool | None*) – In DC mode, take into account transformer ratio. Used only for DC load flows, to include ratios in the equation system.
- **countries_to_balance** (*Sequence[str] | None*) – List of countries participating to slack distribution. Used only if distributed_slack is `True`.
- **component_mode** (*pypowsybl._pypowsybl.ComponentMode | None*) – Defines which network components should be computed. Use `MAIN_SYNCHRONOUS` to computes flows only on the main synchronous component, `MAIN_CONNECTED` to computes flows only on the main connected component, or prefer `ALL_CONNECTED` for a run on all connected components.
- **connected_component_mode** (*pypowsybl._pypowsybl.ConnectedComponentMode | None*) – Deprecated, use parameter `component_mode` (`MAIN` corresponds to component_mode = `MAIN_CONNECTED`, `ALL` to component_mode = `ALL_CONNECTED`).
- **hvdc_ac_emulation** (*bool | None*) – Enable AC emulation of HVDC links.
- **dc_power_factor** (*float | None*) – Power factor used to convert current limits into active power limits in DC calculations.
- **dc** (*bool | None*) – Defines if you want to run an AC power flow (false) or a DC power flow (true).
- **provider_parameters** (*Dict[str, str] | None*) – Define parameters linked to the loadflow provider the names of the existing parameters can be found with method `get_provider_parameters_names`

Some enum classes are used in parameters:

VoltageInitMode
ComponentMode
ConnectedComponentMode

continues on next page

Table 28 – continued from previous page

BalanceType	

pypowsybl.loadflow.VoltageInitMode

class VoltageInitMode(**args*, ***kwargs*)

pypowsybl.loadflow.ComponentMode

class ComponentMode(**args*, ***kwargs*)

pypowsybl.loadflow.ConnectedComponentMode

class ConnectedComponentMode(**args*, ***kwargs*)

pypowsybl.loadflow.BalanceType

class BalanceType(**args*, ***kwargs*)

Results

The loadflow result is actually a list of results, one for each component of the network:

ComponentResult	Loadflow result for one synchronous component of the network.

pypowsybl.loadflow.ComponentResult

class ComponentResult(*res*)

Loadflow result for one synchronous component of the network.

Parameters
res (*pypowsybl._pypowsybl.LoadFlowComponentResult*)

property status: pypowsybl._pypowsybl.LoadFlowComponentStatus

Status of the loadflow for this component.

property status_text: str

Status text of the loadflow for this component.

property connected_component_num: int

Number of the connected component.

property synchronous_component_num: int

Number of the synchronous component.

property iteration_count: int

The number of iterations performed by the loadflow.

property reference_bus_id: str

ID of the (angle) reference bus used for this component.

property slack_bus_results: List[*SlackBusResult*]

Slack bus results for this component.

property distributed_active_power: float

Active power distributed from slack bus to other buses during the loadflow

Some classes and enum classes are used in results:

SlackBusResult	Result for one slack bus of a synchronous component.

pypowsybl.loadflow.SlackBusResult

class SlackBusResult(*res*)

Result for one slack bus of a synchronous component.

Methods

__init__(res)	

Attributes

active_power_mismatch	Slack bus active power mismatch (MW).
id	Slack bus ID.

Parameters

res (*pypowsybl._pypowsybl.SlackBusResult*)

ComponentStatus

pypowsybl.loadflow.ComponentStatus

class ComponentStatus(**args*, ***kwargs*)

Parameters to validate loadflow

The validation of a loadflow can be customized using loadflow validation parameters.

ValidationParameters	Parameters for a loadflow validation.

pypowsybl.loadflow.ValidationParameters

class ValidationParameters(*threshold=None*, *verbose=None*, *loadflow_name=None*, *epsilon_x=None*, *apply_reactance_correction=None*, *loadflow_parameters=None*, *ok_missing_values=None*, *no_requirement_if_reactive_bound_inversion=None*, *compare_results=None*, *check_main_component_only=None*, *no_requirement_if_setpoint_outside_power_bounds=None*)

Parameters for a loadflow validation.

All parameters are first read from you configuration file, then overridden with the constructor arguments.

Parameters

- **threshold** (*float | None*) – Define the margin used for values comparison. The default value is `0`.
- **verbose** (*bool | None*) – Define whether the load flow validation should run in verbose or quiet mode.
- **loadflow_name** (*str | None*) – Implementation name to use for running the load flow.
- **epsilon_x** (*float | None*) – Value used to correct the reactance in flows validation. The default value is `0.1`.
- **apply_reactance_correction** (*bool | None*) – Define whether small reactance values have to be fixed to epsilon_x or not. The default value is `False`.
- **loadflow_parameters** (*Parameters | None*) – Parameters that are common to loadflow and loadflow validation.
- **ok_missing_values** (*bool | None*) – Define whether the validation checks fail if some parameters of connected components have NaN values or not. The default value is `False`.
- **no_requirement_if_reactive_bound_inversion** (*bool | None*) – Define whether the validation checks fail if there is a reactive bounds inversion (maxQ < minQ) or not. The default value is `False`.
- **compare_results** (*bool | None*) – Should be set to `True` to compare the results of 2 validations, i.e. print output files with data of both ones. The default value is `False`.
- **check_main_component_only** (*bool | None*) – Define whether the validation checks are done only on the equiments in the main connected component or in all components. The default value is `True`.
- **no_requirement_if_setpoint_outside_power_bounds** (*bool | None*) – Define whether the validation checks fail if there is a setpoint outside the active power bounds (targetP < minP or targetP > maxP) or not. The default value is `False`.

Methods

`__init__`([threshold, verbose, ...])	
`to_c_parameters`()	

Validate loadflow results

The following method allows to check the consistency of a network with AC loadflow equations.

run_validation	Checks that the network data are consistent with AC loadflow equations.
ValidationResult	The result of a loadflow validation.

pypowsybl.loadflow.run_validation

run_validation(*network*, *validation_types=None*, *validation_parameters=None*)

Checks that the network data are consistent with AC loadflow equations.

Parameters

- **network** (*Network*) – The network to be checked.

- **validation_types** (*List[pypowsybl._pypowsybl.ValidationType] | None*) – The types of data to be checked. If None, all types will be checked.
- **validation_parameters** (ValidationParameters | *None*) – The parameters to run the validation with.

Returns
The validation result.

Return type
ValidationResult

pypowsybl.loadflow.ValidationResult

class ValidationResult(*branch_flows*, *buses*, *generators*, *svcs*, *shunts*, *twts*, *t3wts*)

The result of a loadflow validation.

Methods

__init__(branch_flows, buses, generators, ...)

Attributes

branch_flows	Validation results for branch flows.
buses	Validation results for buses.
generators	Validation results for generators.
shunts	Validation results for shunts.
svcs	Validation results for SVCs.
t3wts	Validation results for three winding transformers.
twts	Validation results for two winding transformers.
valid	True if all checked data is valid.

Parameters

- **branch_flows** (*DataFrame | None*)
- **buses** (*DataFrame | None*)
- **generators** (*DataFrame | None*)
- **svcs** (*DataFrame | None*)
- **shunts** (*DataFrame | None*)
- **twts** (*DataFrame | None*)
- **t3wts** (*DataFrame | None*)

3.1.3 Remedial action optimizer (RAO)

The RAO module allows to optimize remedial actions for electrical power systems (coordinated capacity calculation, local security analysis and coordinated security analysis).

Run a RAO

You can run a RAO using the following methods:

Parameters

The execution of the RAO can be customized using file parameters.

Parameters	

pypowsybl.rao.Parameters

class Parameters(*objective_function_parameters=None*, *range_action_optimization_parameters=None*, *topo_optimization_parameters=None*, *not_optimized_cnecs_parameters=None*, *search_tree_parameters=None*, *fast_rao_parameters=None*, *provider_parameters=None*)

Methods

`__init__`([objective_function_parameters, ...])	
`from_buffer_source`(parameters_source)	
`from_file_source`(parameters_file)	
`serialize`(output_file)	
`serialize_to_binary_buffer`()	
`to_json`()	

Parameters

- **objective_function_parameters** (*ObjectiveFunctionParameters | None*)
- **range_action_optimization_parameters** (*RangeActionOptimizationParameters | None*)
- **topo_optimization_parameters** (*TopoOptimizationParameters | None*)
- **not_optimized_cnecs_parameters** (*NotOptimizedCnecsParameters | None*)
- **search_tree_parameters** (*RaoSearchTreeParameters | None*)
- **fast_rao_parameters** (*FastRaoParameters | None*)
- **provider_parameters** (*Dict[str, str] | None*)

Results

When the RAO is completed, you can inspect its results:

RaoResult	The result of a rao

pypowsybl.rao.RaoResult

class RaoResult(*handle_result*, *handle_crac*)

The result of a rao

Methods

`__init__`(handle_result, handle_crac)	
`from_buffer_source`(crac, result_source)	
`from_file_source`(crac, result_file)	
`get_angle_cnec_results`()	
`get_cost_results`()	
`get_flow_cnec_results`()	
`get_network_action_results`()	
`get_pst_range_action_results`()	
`get_range_action_results`()	
`get_remedial_action_results`()	
`get_virtual_cost_names`()	
`get_virtual_cost_results`(virtual_cost_name)	
`get_voltage_cnec_results`()	
`serialize`(output_file)	Serialize result to file
`serialize_to_binary_buffer`()	Serialize result to BytesIO
`status`()	
`to_json`()	

Parameters

- **handle_result** (*pypowsybl._pypowsybl.JavaHandle*)
- **handle_crac** (*pypowsybl._pypowsybl.JavaHandle*)

3.1.4 Security analysis

The security analysis module allows to simulate contingencies (the loss of a line or a generator, for example) in a "batch" mode, in AC mode.

Run a security analysis

You can run a security analysis using the following methods:

create_analysis	Creates a security analysis objet, which can be used to run a security analysis on a network
SecurityAnalysis.run_ac	Runs an AC security analysis.
SecurityAnalysis.run_dc	Runs a DC security analysis.
set_default_provider	Set the default security analysis provider.
get_default_provider	Get the current default security analysis provider.
get_provider_names	Get list of supported provider names

pypowsybl.security.create_analysis

`create_analysis`()

Creates a security analysis objet, which can be used to run a security analysis on a network

Examples

```
>>> analysis = pypowsybl.security.create_analysis()
>>> analysis.add_single_element_contingencies(['line 1', 'line 2'])
>>> res = analysis.run_ac(network)
```

Returns
A security analysis object, which allows to run a security analysis on a network.

Return type
SecurityAnalysis

pypowsybl.security.SecurityAnalysis.run_ac

`SecurityAnalysis.`**`run_ac`**(*network*, *parameters=None*, *provider=''*, *reporter=None*, *report_node=None*)

Runs an AC security analysis.

Parameters

- **network** (Network) – Network on which the security analysis will be computed
- **parameters** (Parameters | Parameters | *None*) – Security analysis parameters
- **provider** (*str*) – Name of the security analysis implementation provider to be used, will use default provider if empty.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Returns
A security analysis result, containing information about violations and monitored elements

Return type
SecurityAnalysisResult

pypowsybl.security.SecurityAnalysis.run_dc

`SecurityAnalysis.`**`run_dc`**(*network*, *parameters=None*, *provider=''*, *reporter=None*, *report_node=None*)

Runs a DC security analysis.

Parameters

- **network** (Network) – Network on which the security analysis will be computed
- **parameters** (Parameters | Parameters | *None*) – Security analysis parameters
- **provider** (*str*) – Name of the security analysis implementation provider to be used, will use default provider if empty.
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Returns
A security analysis result, containing information about violations and monitored elements

Return type
SecurityAnalysisResult

pypowsybl.security.set_default_provider

`set_default_provider`(*provider*)

Set the default security analysis provider.

Parameters
provider (*str*) – name of the default security analysis provider to set

Return type
None

pypowsybl.security.get_default_provider

get_default_provider()

Get the current default security analysis provider.

Returns
the name of the current default security analysis provider

Return type
str

pypowsybl.security.get_provider_names

get_provider_names()

Get list of supported provider names

Returns
the list of supported provider names

Return type
List[str]

Parameters

The execution of the security analysis can be customized using security analysis parameters.

Parameters	Parameters for a security analysis execution.
IncreasedViolationsParameters	Parameters which define what violations should be considered as "increased" between N and post-contingency situations
get_provider_parameters_names	Get list of parameters for the specified security analysis provider.

pypowsybl.security.Parameters

class Parameters(*load_flow_parameters=None*, *increased_violations_parameters=None*, *provider_parameters=None*)

Parameters for a security analysis execution.

All parameters are first read from you configuration file, then overridden with the constructor arguments.

Please note that security analysis providers may not honor all parameters, according to their capabilities. For example, some providers will not be able to simulate the voltage control of shunt compensators, etc. The exact behaviour of some parameters may also depend on your security analysis provider. Please check the documentation of your provider for that information.

Parameters

- **load_flow_parameters** (Parameters | *None*) – parameters that are common to load-flow and security analysis

- **increased_violations_parameters** (IncreasedViolationsParameters | *None*) – Define what violations should be considered increased between N and contingency situations
- **provider_parameters** (*Dict[str, str] | None*) – Define parameters linked to the security analysis provider the names of the existing parameters can be found with method `get_provider_parameters_names`

Methods

`__init__`([load_flow_parameters, ...])	

Attributes

`increased_violations`	Define what violations should be considered increased between N and post-contingency situations

pypowsybl.security.IncreasedViolationsParameters

class IncreasedViolationsParameters(*flow_proportional_threshold*, *low_voltage_proportional_threshold*, *low_voltage_absolute_threshold*, *high_voltage_proportional_threshold*, *high_voltage_absolute_threshold*)

Parameters which define what violations should be considered as "increased" between N and post-contingency situations

Parameters

- **flow_proportional_threshold** (*float*) – for current and flow violations, if equal to 0.1, the violations which value have increased of more than 10% between N and post-contingency are considered "increased"
- **low_voltage_proportional_threshold** (*float*) – for low voltage violations, if equal to 0.1, the violations which value have reduced of more than 10% between N and post-contingency are considered "increased"
- **low_voltage_absolute_threshold** (*float*) – for low voltage violations, if equal to 1, the violations which value have reduced of more than 1 kV between N and post-contingency are considered "increased"
- **high_voltage_proportional_threshold** (*float*) – for high voltage violations, if equal to 0.1, the violations which value have increased of more than 10% between N and post-contingency are considered "increased"
- **high_voltage_absolute_threshold** (*float*) – for high voltage violations, if equal to 1, the violations which value have increased of more than 1 kV between N and post-contingency are considered "increased"

Methods

`__init__`(flow_proportional_threshold, ...)	

pypowsybl.security.get_provider_parameters_names

get_provider_parameters_names(*provider=''*)

Get list of parameters for the specified security analysis provider.

If not specified the provider will be the default one.

Returns
the list of provider's parameters

Parameters
provider (*str*)

Return type
List[str]

Define contingencies and monitored elements

You can define contingencies to be simulated, as well as network elements to be monitored, with the following methods:

SecurityAnalysis.add_single_element_continge	Add one N-1 contingency.
SecurityAnalysis.add_multiple_elements_conti	Add one N-K contingency.
SecurityAnalysis.add_single_element_continge	Add multiple N-1 contingencies.
SecurityAnalysis.add_monitored_elements	Add elements to be monitored by the security analysis.
SecurityAnalysis.add_precontingency_monitore	Add elements to be monitored by the security analysis on precontingency state.
SecurityAnalysis.add_postcontingency_monito	Add elements to be monitored by the security analysis for specific contingencies.

pypowsybl.security.SecurityAnalysis.add_single_element_contingency

SecurityAnalysis.**add_single_element_contingency**(*element_id*, *contingency_id=None*)

Add one N-1 contingency.

Parameters

- **element_id** (*str*) – The ID of the lost network element.
- **contingency_id** (*str | None*) – The ID of the contingency. If None, element_id will be used.

Return type
None

pypowsybl.security.SecurityAnalysis.add_multiple_elements_contingency

SecurityAnalysis.**add_multiple_elements_contingency**(*elements_ids*, *contingency_id*)

Add one N-K contingency.

Parameters

- **elements_ids** (*List[str]*) – The ID of the lost network elements.
- **contingency_id** (*str*) – The ID of the contingency.

Return type
None

pypowsybl.security.SecurityAnalysis.add_single_element_contingencies

SecurityAnalysis.**add_single_element_contingencies**(*elements_ids*, *contingency_id_provider=None*)

Add multiple N-1 contingencies.

Parameters

- **elements_ids** (*List[str]*) – A list of network elements. One N- 1 contingency will be added for each element of the list.
- **contingency_id_provider** (*Callable[[str], str] | None*) – A callable which maps elements IDs to a contingency ID. If None, the element ID will be used as the contingency ID for each N-1 contingency.

Return type
None

pypowsybl.security.SecurityAnalysis.add_monitored_elements

SecurityAnalysis.**add_monitored_elements**(*contingency_context_type=pypowsybl._pypowsybl.ContingencyContextType.ALL*, *contingency_ids=None*, *branch_ids=None*, *voltage_level_ids=None*, *three_windings_transformer_ids=None*)

Add elements to be monitored by the security analysis. The security analysis result will provide additional information for those elements, like the power and current values.

Parameters

- **contingency_context_type** (*pypowsybl._pypowsybl.ContingencyContextType*) – Defines if the elements should be monitored for all state, only N situation or only specific contingencies
- **contingency_ids** (*List[str] | str | None*) – list of contingencies for which we want to monitor additional elements
- **branch_ids** (*List[str] | None*) – list of branches to be monitored
- **voltage_level_ids** (*List[str] | None*) – list of voltage levels to be monitored
- **three_windings_transformer_ids** (*List[str] | None*) – list of 3 winding transformers to be monitored

Return type
None

pypowsybl.security.SecurityAnalysis.add_precontingency_monitored_elements

SecurityAnalysis.**add_precontingency_monitored_elements**(*branch_ids=None*, *voltage_level_ids=None*, *three_windings_transformer_ids=None*)

Add elements to be monitored by the security analysis on precontingency state. The security analysis result will provide additional information for those elements, like the power and current values.

Parameters

- **branch_ids** (*List[str] | None*) – list of branches to be monitored
- **voltage_level_ids** (*List[str] | None*) – list of voltage levels to be monitored
- **three_windings_transformer_ids** (*List[str] | None*) – list of 3 winding transformers to be monitored

Return type
None

pypowsybl.security.SecurityAnalysis.add_postcontingency_monitored_elements

SecurityAnalysis.**add_postcontingency_monitored_elements**(*contingency_ids*, *branch_ids=None*, *voltage_level_ids=None*, *three_windings_transformer_ids=None*)

Add elements to be monitored by the security analysis for specific contingencies. The security analysis result will provide additional information for those elements, like the power and current values.

Parameters

- **contingency_ids** (*List[str] | str*) – list of contingencies for which we want to monitor additional elements
- **branch_ids** (*List[str] | None*) – list of branches to be monitored
- **voltage_level_ids** (*List[str] | None*) – list of voltage levels to be monitored
- **three_windings_transformer_ids** (*List[str] | None*) – list of 3 winding transformers to be monitored

Return type
None

Define operator strategies and remedial actions

You can define operator strategies and remedial actions with the following methods:

SecurityAnalysis.add_load_active_power_actio	Add a load action, modifying the load active power
SecurityAnalysis.add_load_reactive_power_act	Add a load action, modifying the load reactive power
*SecurityAnalysis.add_generator_active_power_*	Add a generator action, modifying the generator active power
SecurityAnalysis.add_switch_action	Add a switch action, modifying the switch open/close status
SecurityAnalysis.add_phase_tap_changer_posit	Add a phase tap changer tap position action, modifying the tap position of the tap changer
SecurityAnalysis.add_ratio_tap_changer_posit	Add a ratio tap changer tap position action, modifying the tap position of the tap changer
SecurityAnalysis.add_shunt_compensator_posit	Add a shunt compensator section action, modifying the section of the shunt compensator
SecurityAnalysis.add_terminals_connection_ac	Add a terminals connection action, connecting/disconnecting one or multiple sides of a network element
SecurityAnalysis.add_operator_strategy	Add an operator strategy to the specified contingency
SecurityAnalysis.add_actions_from_json_file	Add any kinds of actions by reading them from a JSON file.
SecurityAnalysis.add_operator_strategies_fro	Add operator strategies by reading them from a JSON file.

pypowsybl.security.SecurityAnalysis.add_load_active_power_action

SecurityAnalysis.**add_load_active_power_action**(*action_id*, *load_id*, *is_relative*, *active_power*)

Add a load action, modifying the load active power

Parameters

- **action_id** (*str*) – unique ID for the action
- **load_id** (*str*) – load identifier
- **is_relative** (*bool*) – whether the active power change specified is absolute, or relative to current load active power
- **active_power** (*float*) – the active power change

Return type
None

pypowsybl.security.SecurityAnalysis.add_load_reactive_power_action

SecurityAnalysis.**add_load_reactive_power_action**(*action_id*, *load_id*, *is_relative*, *reactive_power*)

Add a load action, modifying the load reactive power

Parameters

- **action_id** (*str*) – unique ID for the action
- **load_id** (*str*) – load identifier
- **is_relative** (*bool*) – whether the reactive power change specified is absolute, or relative to current load reactive power
- **reactive_power** (*float*) – the reactive power change

Return type
None

pypowsybl.security.SecurityAnalysis.add_generator_active_power_action

SecurityAnalysis.**add_generator_active_power_action**(*action_id*, *generator_id*, *is_relative*, *active_power*)

Add a generator action, modifying the generator active power

Parameters

- **action_id** (*str*) – unique ID for the action
- **generator_id** (*str*) – generator identifier
- **is_relative** (*bool*) – whether the active power change specified is absolute, or relative to current generator active power
- **active_power** (*float*) – the active power change

Return type
None

pypowsybl.security.SecurityAnalysis.add_switch_action

SecurityAnalysis.**add_switch_action**(*action_id*, *switch_id*, *open*)

Add a switch action, modifying the switch open/close status

Parameters

- **action_id** (*str*) – unique ID for the action
- **switch_id** (*str*) – switch identifier
- **open** (*bool*) – True to open the switch, False to close

Return type
None

pypowsybl.security.SecurityAnalysis.add_phase_tap_changer_position_action

SecurityAnalysis.**add_phase_tap_changer_position_action**(*action_id*, *transformer_id*, *is_relative*, *tap_position*, *side=pypowsybl._pypowsybl.Side.NONE*)

Add a phase tap changer tap position action, modifying the tap position of the tap changer

Parameters

- **action_id** (*str*) – unique ID for the action
- **transformer_id** (*str*) – transformer identifier
- **is_relative** (*bool*) – True means the provided tap_position will be added to the current tap position, False means the provided tap_position will replace the previous one.
- **tap_position** (*int*) – The tap position (either a delta if is_relative is true, or the final value if is_relative if false)
- **side** (*pypowsybl._pypowsybl.Side*) – Side of the tap changer (for three windings transformers)

Return type
None

pypowsybl.security.SecurityAnalysis.add_ratio_tap_changer_position_action

SecurityAnalysis.**add_ratio_tap_changer_position_action**(*action_id*, *transformer_id*, *is_relative*, *tap_position*, *side=pypowsybl._pypowsybl.Side.NONE*)

Add a ratio tap changer tap position action, modifying the tap position of the tap changer

Parameters

- **action_id** (*str*) – unique ID for the action
- **transformer_id** (*str*) – transformer identifier
- **is_relative** (*bool*) – True means the provided tap_position will be added to the current tap position, False means the provide tap_position will replace the previous one.
- **tap_position** (*int*) – The tap position (either a delta if is_relative is true, or the final value if is_relative if false)
- **side** (*pypowsybl._pypowsybl.Side*) – Side of the tap changer (for three windings transformers)

Return type
None

pypowsybl.security.SecurityAnalysis.add_shunt_compensator_position_action

SecurityAnalysis.**add_shunt_compensator_position_action**(*action_id*, *shunt_id*, *section*)

Add a shunt compensator section action, modifying the section of the shunt compensator

Parameters

- **action_id** (*str*) – unique ID for the action
- **shunt_id** (*str*) – transformer identifier
- **section** (*int*) – The new section of the shunt compensator

Return type
None

pypowsybl.security.SecurityAnalysis.add_terminals_connection_action

SecurityAnalysis.**add_terminals_connection_action**(*action_id*, *element_id*, *side=pypowsybl._pypowsybl.Side.NONE*, *opening=True*)

Add a terminals connection action, connecting/disconnecting one or multiple sides of a network element

Parameters

- **action_id** (*str*) – unique ID for the action
- **element_id** (*str*) – network element identifier
- **side** (*pypowsybl._pypowsybl.Side*) – The side of the element to modify (all if side=None)
- **opening** (*bool*) – True to open the terminals, False otherwise

Return type
None

pypowsybl.security.SecurityAnalysis.add_operator_strategy

SecurityAnalysis.**add_operator_strategy**(*operator_strategy_id*, *contingency_id*, *action_ids*, *condition_type=pypowsybl._pypowsybl.ConditionType.TRUE_CONDITION*, *violation_subject_ids=None*, *violation_types=None*)

Add an operator strategy to the specified contingency

Parameters

- **operator_strategy_id** (*str*) – unique ID for the operator strategy
- **contingency_id** (*str*) – the contingency on which the operator strategy applies
- **action_ids** (*List[str]*) – the list of actions to be applied as part of the strategy
- **condition_type** (*pypowsybl._pypowsybl.ConditionType*) – the type of condition
- **violation_subject_ids** (*List[str] | None*) – identifiers of network elements monitored to apply the operator strategy
- **violation_types** (*List[pypowsybl._pypowsybl.ViolationType] | None*) – type of violations to consider to apply the operator strategy

Return type
None

pypowsybl.security.SecurityAnalysis.add_actions_from_json_file

SecurityAnalysis.**add_actions_from_json_file**(*path_to_json_file*)

Add any kinds of actions by reading them from a JSON file.

Parameters
path_to_json_file (*str*) – the path to the JSON file in which we extract the actions' data.

Return type
None

pypowsybl.security.SecurityAnalysis.add_operator_strategies_from_json_file

SecurityAnalysis.**add_operator_strategies_from_json_file**(*path_to_json_file*)

Add operator strategies by reading them from a JSON file.

Parameters
path_to_json_file (*str*) – the path to the JSON file in which we extract the operator strategies' data.

Return type
None

Define limit reductions

You can define limit reductions using :

SecurityAnalysis.add_limit_reductions	Add limit reductions to the security analysis.

pypowsybl.security.SecurityAnalysis.add_limit_reductions

SecurityAnalysis.**add_limit_reductions**(*df=None*, ***kwargs*)

Add limit reductions to the security analysis. If two reductions can be applied to the same limit, the last one in addition order is the one applied.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **limit_type**: the type of limits to be reduced (only CURRENT is currently supported in OpenLoadFlow).

- **permanent**: whether the limit reduction should be applied on permanent limits or not.
- **temporary**: whether the limit reduction should be applied on temporary limits or not.
- **value**: the value of the limit reduction (must be in [0, 1]).
- **contingency_context** (optional, only ALL implemented in OpenLoadFlow): defines if the reduction should be applied on all state, only N situation or only specific contingencies.
- **min_temporary_duration** (optional): if temporary is True, the minimum acceptable duration of the temporary limits affected by this reduction (in seconds).
- **max_temporary_duration** (optional): if temporary is True, the maximum acceptable duration of the temporary limits affected by this reduction (in seconds).
- **country** (optional): the country on which the limit reduction should be applied.
- **min_voltage** (optional): the minimum voltage level on which the limit reduction should be applied.
- **max_voltage** (optional): the maximum voltage level on which the limit reduction should be applied.

Examples

Adding a limit reduction of 0.8 on all current limits:

```
security_analysis.add_limit_reductions(limit_type='CURRENT', permanent=True,␣
↪temporary=True, value=0.8)
```

Adding a limit reduction of 0.9 on all temporary current limits with minimal acceptable duration of 300s, on all network elements in France with nominal voltage between 90 and 225kV :

```
security_analysis.add_limit_reductions(limit_type='CURRENT', permanent=False,␣
↪temporary=True, value=0.9, min_temporary_duration=300, country='FR', min_
↪voltage=90, max_voltage=225)
```

Results

When the security analysis is completed, you can inspect its results:

SecurityAnalysisResult	The result of a security analysis.
SecurityAnalysisResult.limit_violations	All limit violations in a dataframe representation.
SecurityAnalysisResult.pre_contingency_result	Result for the pre-contingency state.
SecurityAnalysisResult.post_contingency_results	Results for the contingencies, as a dictionary contingency ID -> result.
SecurityAnalysisResult.operator_strategy_results	Results for the operator strategies, as a dictionary operator strategy ID -> result.
SecurityAnalysisResult.find_post_contingency_result	Result for the specified contingency.
SecurityAnalysisResult.find_operator_strategy_results	Result for the specified operator strategy
SecurityAnalysisResult.branch_results	Results (P, Q, I) for monitored branches.
SecurityAnalysisResult.bus_results	Bus results (voltage angle and magnitude) for monitored voltage levels.
SecurityAnalysisResult.three_windings_transformer_results	Results (P, Q, I) for monitored three winding transformers.

continues on next page

Table 52 – continued from previous page

SecurityAnalysisResult.export_to_json	Add the security analysis' output to the specified path in a JSON file.

pypowsybl.security.SecurityAnalysisResult

class SecurityAnalysisResult(*handle*)

The result of a security analysis.

Methods

__init__(handle)	
export_to_json(path)	Add the security analysis' output to the specified path in a JSON file.
find_operator_strategy_results(...)	Result for the specified operator strategy
find_post_contingency_result(contingency_id)	Result for the specified contingency.
get_table()	

Attributes

branch_results	Results (P, Q, I) for monitored branches.
bus_results	Bus results (voltage angle and magnitude) for monitored voltage levels.
limit_violations	All limit violations in a dataframe representation.
operator_strategy_results	Results for the operator strategies, as a dictionary operator strategy ID -> result.
post_contingency_results	Results for the contingencies, as a dictionary contingency ID -> result.
pre_contingency_result	Result for the pre-contingency state.
three_windings_transformer_results	Results (P, Q, I) for monitored three winding transformers.

Parameters

handle (*pypowsybl._pypowsybl.JavaHandle*)

pypowsybl.security.SecurityAnalysisResult.limit_violations

property SecurityAnalysisResult.**limit_violations: DataFrame**

All limit violations in a dataframe representation.

pypowsybl.security.SecurityAnalysisResult.pre_contingency_result

property SecurityAnalysisResult.**pre_contingency_result: pypowsybl._pypowsybl.PreContingencyResult**

Result for the pre-contingency state.

pypowsybl.security.SecurityAnalysisResult.post_contingency_results

property SecurityAnalysisResult.**post_contingency_results**: Dict[str, pypowsybl._pypowsybl.PostContingencyResult]

Results for the contingencies, as a dictionary contingency ID -> result.

pypowsybl.security.SecurityAnalysisResult.operator_strategy_results

property SecurityAnalysisResult.**operator_strategy_results**: Dict[str, pypowsybl._pypowsybl.OperatorStrategyResult]

Results for the operator strategies, as a dictionary operator strategy ID -> result.

pypowsybl.security.SecurityAnalysisResult.find_post_contingency_result

SecurityAnalysisResult.**find_post_contingency_result**(*contingency_id*)

Result for the specified contingency.

Returns
Result for the specified contingency.

Parameters
contingency_id (*str*)

Return type
pypowsybl._pypowsybl.PostContingencyResult

pypowsybl.security.SecurityAnalysisResult.find_operator_strategy_results

SecurityAnalysisResult.**find_operator_strategy_results**(*operator_strategy_id*)

Result for the specified operator strategy

Returns
Result for the specified operator strategy.

Parameters
operator_strategy_id (*str*)

Return type
pypowsybl._pypowsybl.OperatorStrategyResult

pypowsybl.security.SecurityAnalysisResult.branch_results

property SecurityAnalysisResult.**branch_results**: DataFrame

Results (P, Q, I) for monitored branches.

pypowsybl.security.SecurityAnalysisResult.bus_results

property SecurityAnalysisResult.**bus_results**: DataFrame

Bus results (voltage angle and magnitude) for monitored voltage levels.

pypowsybl.security.SecurityAnalysisResult.three_windings_transformer_results

property SecurityAnalysisResult.**three_windings_transformer_results**: DataFrame

Results (P, Q, I) for monitored three winding transformers.

pypowsybl.security.SecurityAnalysisResult.export_to_json

SecurityAnalysisResult.**export_to_json**(*path*)

Add the security analysis' output to the specified path in a JSON file.

Args :

path : the path where we ant the JSON file to be written after the security analysis.

Parameters

path (*str*)

Return type

None

3.1.5 Sensitivity analysis

The sensitivity analysis module allows to compute the impact of various variations (typically, generation variations), on other physical values on the network (typically, active power flows or currents on lines).

Run a sensitivity analysis

You can run an AC or DC security analysis using the following methods:

create_ac_analysis	Creates a new AC sensitivity analysis.
AcSensitivityAnalysis.run	Runs the sensitivity analysis.
create_dc_analysis	Creates a new DC sensitivity analysis.
DcSensitivityAnalysis.run	Runs the sensitivity analysis
set_default_provider	Set the default sensitivity analysis provider
get_default_provider	Get the current default sensitivity analysis provider.
get_provider_names	Get list of supported provider names

pypowsybl.sensitivity.create_ac_analysis

create_ac_analysis()

Creates a new AC sensitivity analysis.

Returns

a new AC sensitivity analysis

Return type

AcSensitivityAnalysis

pypowsybl.sensitivity.AcSensitivityAnalysis.run

AcSensitivityAnalysis.**run**(*network*, *parameters=None*, *provider=''*, *reporter=None*, *report_node=None*)

Runs the sensitivity analysis.

Parameters

- **network** (Network) – The network
- **parameters** (Parameters | Parameters | *None*) – The sensitivity parameters
- **provider** (*str*) – Name of the sensitivity analysis provider
- **reporter** (*ReportNode* | *None*) – deprecated, use report_node instead

- **report_node** (*ReportNode | None*) – The reporter to be used to create an execution report, default is None (no report)

Returns
a sensitivity analysis result

Return type
AcSensitivityAnalysisResult

pypowsybl.sensitivity.create_dc_analysis

create_dc_analysis()

Creates a new DC sensitivity analysis.

Returns
a new DC sensitivity analysis

Return type
DcSensitivityAnalysis

pypowsybl.sensitivity.DcSensitivityAnalysis.run

DcSensitivityAnalysis.**run**(*network*, *parameters=None*, *provider=''*, *reporter=None*, *report_node=None*)

Runs the sensitivity analysis

Parameters

- **network** (Network) – The network
- **parameters** (Parameters | Parameters | *None*) – The sensitivity parameters
- **provider** (*str*) – Name of the sensitivity analysis provider
- **reporter** (*ReportNode | None*) – deprecated, use report_node instead
- **report_node** (*ReportNode | None*) – the reporter to be used to create an execution report, default is None (no report)

Returns
a sensitivity analysis result

Return type
DcSensitivityAnalysisResult

pypowsybl.sensitivity.set_default_provider

set_default_provider(*provider*)

Set the default sensitivity analysis provider

Parameters
provider (*str*) – name of the default sensitivity analysis provider to set

Return type
None

pypowsybl.sensitivity.get_default_provider

get_default_provider()

Get the current default sensitivity analysis provider.

Returns
the name of the current default sensitivity analysis provider

Return type
str

pypowsybl.sensitivity.get_provider_names

`get_provider_names()`

Get list of supported provider names

Returns
the list of supported provider names

Return type
List[str]

Parameters

The execution of the sensitivity analysis can be customized using sensitivity analysis parameters.

Parameters	Parameters for a sensitivity analysis execution.
get_provider_parameters_names	Get list of parameters for the specified sensitivity analysis provider.

pypowsybl.sensitivity.Parameters

`class Parameters`(*load_flow_parameters=None*, *provider_parameters=None*, *flow_flow_sensitivity_value_threshold=None*, *voltage_voltage_sensitivity_value_threshold=None*, *flow_voltage_sensitivity_value_threshold=None*, *angle_flow_sensitivity_value_threshold=None*)

Parameters for a sensitivity analysis execution.

All parameters are first read from you configuration file, then overridden with the constructor arguments.

Please note that sensitivity providers may not honor all parameters, according to their capabilities. The exact behaviour of some parameters may also depend on your sensitivity provider. Please check the documentation of your provider for that information.

Parameters

- **load_flow_parameters** (*Parameters | None*) – parameters that are common to load-flow and sensitivity analysis
- **provider_parameters** (*Dict[str, str] | None*) – Define parameters linked to the sensitivity analysis provider the names of the existing parameters can be found with method `get_provider_parameters_names`
- **flow_flow_sensitivity_value_threshold** (*float | None*)
- **voltage_voltage_sensitivity_value_threshold** (*float | None*)
- **flow_voltage_sensitivity_value_threshold** (*float | None*)
- **angle_flow_sensitivity_value_threshold** (*float | None*)

Methods

`__init__`([load_flow_parameters, ...])	

pypowsybl.sensitivity.get_provider_parameters_names

get_provider_parameters_names(*provider=''*)

Get list of parameters for the specified sensitivity analysis provider.

If not specified the provider will be the default one.

Returns
the list of provider's parameters

Parameters
provider (*str*)

Return type
List[str]

Contingencies definition

SensitivityAnalysis.add_single_element_contingency	Add one N-1 contingency.
SensitivityAnalysis.add_multiple_elements_contingency	Add one N-K contingency.
SensitivityAnalysis.add_single_element_contingencies	Add multiple N-1 contingencies.

pypowsybl.sensitivity.SensitivityAnalysis.add_single_element_contingency

SensitivityAnalysis.**add_single_element_contingency**(*element_id*, *contingency_id=None*)

Add one N-1 contingency.

Parameters

- **element_id** (*str*) – The ID of the lost network element.
- **contingency_id** (*str | None*) – The ID of the contingency. If None, element_id will be used.

Return type
None

pypowsybl.sensitivity.SensitivityAnalysis.add_multiple_elements_contingency

SensitivityAnalysis.**add_multiple_elements_contingency**(*elements_ids*, *contingency_id*)

Add one N-K contingency.

Parameters

- **elements_ids** (*List[str]*) – The ID of the lost network elements.
- **contingency_id** (*str*) – The ID of the contingency.

Return type
None

pypowsybl.sensitivity.SensitivityAnalysis.add_single_element_contingencies

SensitivityAnalysis.**add_single_element_contingencies**(*elements_ids*, *contingency_id_provider=None*)

Add multiple N-1 contingencies.

Parameters

- **elements_ids** (*List[str]*) – A list of network elements. One N- 1 contingency will be added for each element of the list.
- **contingency_id_provider** (*Callable[[str], str] | None*) – A callable which maps elements IDs to a contingency ID. If None, the element ID will be used as the contingency ID for each N-1 contingency.

Return type

None

Sensitivities definition

You can either define the sensitivities you want to compute by defining individual elements variations, or by defining zones.

In AC mode, you can define voltage sensitivities, in addition to flows sensitivities.

SensitivityAnalysis.add_branch_flow_factor_matrix	Defines branch active power flow factor matrix, with a list of branches IDs and a list of variables.
SensitivityAnalysis.add_precontingency_branch_flow_factor_matri	Defines branch active power flow factor matrix for the base case, with a list of branches IDs and a list of variables.
SensitivityAnalysis.add_postcontingency_branch_flow_factor_matr	Defines branch active power flow factor matrix for specific post contingencies states, with a list of branches IDs and a list of variables.
AcSensitivityAnalysis.set_bus_voltage_factor_matrix	
SensitivityAnalysis.set_zones	Define zones that will be used in branch flow factor matrix.

pypowsybl.sensitivity.SensitivityAnalysis.add_branch_flow_factor_matrix

SensitivityAnalysis.**add_branch_flow_factor_matrix**(*branches_ids*, *variables_ids*, *matrix_id='default'*)

Defines branch active power flow factor matrix, with a list of branches IDs and a list of variables.

A variable could be:

- a network element ID: injections, PSTs, boundary lines and HVDC lines are supported
- a zone ID
- a couple of zone ID to define a transfer between 2 zones

Parameters

- **branches_ids** (*List[str]*) – IDs of branches for which active power flow sensitivities should be computed
- **variables_ids** (*List[str]*) – variables which may impact branch flows,to which we should compute sensitivities
- **matrix_id** (*str*) – The matrix unique identifier, to be used to retrieve the sensibility value

Return type
None

pypowsybl.sensitivity.SensitivityAnalysis.add_precontingency_branch_flow_factor_matrix

SensitivityAnalysis.**add_precontingency_branch_flow_factor_matrix**(*branches_ids*, *variables_ids*, *matrix_id='default'*)

Defines branch active power flow factor matrix for the base case, with a list of branches IDs and a list of variables.

A variable could be:

- a network element ID: injections, PSTs, boundary lines and HVDC lines are supported
- a zone ID
- a couple of zone ID to define a transfer between 2 zones

Parameters

- **branches_ids** (*List[str]*) – IDs of branches for which active power flow sensitivities should be computed
- **variables_ids** (*List[str]*) – variables which may impact branch flows,to which we should compute sensitivities
- **matrix_id** (*str*) – The matrix unique identifier, to be used to retrieve the sensibility value

Return type
None

pypowsybl.sensitivity.SensitivityAnalysis.add_postcontingency_branch_flow_factor_matrix

SensitivityAnalysis.**add_postcontingency_branch_flow_factor_matrix**(*branches_ids*, *variables_ids*, *contingencies_ids*, *matrix_id='default'*)

Defines branch active power flow factor matrix for specific post contingencies states, with a list of branches IDs and a list of variables.

A variable could be:

- a network element ID: injections, PSTs, boundary lines and HVDC lines are supported
- a zone ID
- a couple of zone ID to define a transfer between 2 zones

Parameters

- **branches_ids** (*List[str]*) – IDs of branches for which active power flow sensitivities should be computed
- **variables_ids** (*List[str]*) – variables which may impact branch flows,to which we should compute sensitivities
- **contingencies_ids** (*List[str]*) – List of the IDs of the contingencies to simulate
- **matrix_id** (*str*) – The matrix unique identifier, to be used to retrieve the sensibility value

Return type
None

pypowsybl.sensitivity.AcSensitivityAnalysis.set_bus_voltage_factor_matrix

AcSensitivityAnalysis.**set_bus_voltage_factor_matrix**(*bus_ids*, *target_voltage_ids*)

Deprecated since version 1.1.0: Use `add_bus_voltage_factor_matrix()` instead.

Defines buses voltage sensitivities to be computed.

Parameters

- **bus_ids** (*List[str]*) – IDs of buses for which voltage sensitivities should be computed
- **target_voltage_ids** (*List[str]*) – IDs of regulating equipments to which we should compute sensitivities

Return type
None

pypowsybl.sensitivity.SensitivityAnalysis.set_zones

SensitivityAnalysis.**set_zones**(*zones*)

Define zones that will be used in branch flow factor matrix.

Parameters
zones (*List[Zone]*) – a list of zones

Return type
None

In order to create, inspect and manipulate zones, you can use the following methods:

create_empty_zone	
create_country_zone	
create_zone_from_injections_and_shift_keys	Create country zone with custom generator name and shift keys :param country: Identifier of the zone :param injection_index: IDs of the injection :param shift_keys: shift keys for the generators
create_zones_from_glsk_file	Create country zones from glsk file for a given datetime :param glsk_file: UCTE glsk file :param instant: time-point at which to select glsk data
Zone	
Zone.id	
Zone.shift_keys_by_injections_ids	
Zone.injections_ids	
Zone.get_shift_key	
Zone.add_injection	
Zone.remove_injection	
Zone.move_injection_to	
ZoneKeyType	

pypowsybl.sensitivity.create_empty_zone

create_empty_zone(*id*)

Parameters
id (*str*)

Return type
Zone

pypowsybl.sensitivity.create_country_zone

create_country_zone(*network*, *country*, *key_type=ZoneKeyType.GENERATOR_TARGET_P*)

Parameters

- **network** (Network)
- **country** (*str*)
- **key_type** (ZoneKeyType)

Return type
Zone

pypowsybl.sensitivity.create_zone_from_injections_and_shift_keys

create_zone_from_injections_and_shift_keys(*id*, *injection_index*, *shift_keys*)

Create country zone with custom generator name and shift keys :param country: Identifier of the zone :param injection_index: IDs of the injection :param shift_keys: shift keys for the generators

Returns
The zone object

Parameters

- **id** (*str*)
- **injection_index** (*List[str]*)
- **shift_keys** (*List[float]*)

Return type
Zone

pypowsybl.sensitivity.create_zones_from_glsk_file

create_zones_from_glsk_file(*network*, *glsk_file*, *instant*)

Create country zones from glsk file for a given datetime :param glsk_file: UCTE glsk file :param instant: time-point at which to select glsk data

Returns
A list of zones created from glsk file

Parameters

- **network** (Network)
- **glsk_file** (*str*)
- **instant** (*datetime*)

Return type
List[Zone]

pypowsybl.sensitivity.Zone

class Zone(*id*, *shift_keys_by_injections_ids=None*)

Methods

`__init__`(id[, shift_keys_by_injections_ids])	
add_injection(id[, key])	
get_shift_key(injection_id)	
move_injection_to(other_zone, id)	
remove_injection(id)	

Attributes

id	
injections_ids	
shift_keys_by_injections_ids	

Parameters

- **id** (*str*)
- **shift_keys_by_injections_ids** (*Dict[str, float] | None*)

pypowsybl.sensitivity.Zone.id

property Zone.**id**: str

pypowsybl.sensitivity.Zone.shift_keys_by_injections_ids

property Zone.**shift_keys_by_injections_ids**: Dict[str, float]

pypowsybl.sensitivity.Zone.injections_ids

property Zone.**injections_ids**: List[str]

pypowsybl.sensitivity.Zone.get_shift_key

Zone.**get_shift_key**(*injection_id*)

Parameters
injection_id (*str*)

Return type
float

pypowsybl.sensitivity.Zone.add_injection

Zone.**add_injection**(*id*, *key=1*)

Parameters

- **id** (*str*)
- **key** (*float*)

Return type
None

pypowsybl.sensitivity.Zone.remove_injection

Zone.**remove_injection**(*id*)

Parameters
id (*str*)

Return type
None

pypowsybl.sensitivity.Zone.move_injection_to

Zone.**move_injection_to**(*other_zone*, *id*)

Parameters

- **other_zone** (Zone)
- **id** (*str*)

Return type
None

pypowsybl.sensitivity.ZoneKeyType

class ZoneKeyType(**values*)

Attributes

GENERATOR_TARGET_P
GENERATOR_MAX_P
LOAD_P0

Results

When the security analysis is completed, you can inspect its results:

DcSensitivityAnalysisResult	Represents the result of a DC sensitivity analysis.
DcSensitivityAnalysisResult.get_branch_flows_sensitivity_matrix	
DcSensitivityAnalysisResult.get_reference_flows	
AcSensitivityAnalysisResult	Represents the result of an AC sensitivity analysis.
AcSensitivityAnalysisResult.get_bus_voltages_sensitivity_matrix	
AcSensitivityAnalysisResult.get_reference_voltages	
SensitivityAnalysisResult	Represents the result of a sensitivity analysis.
SensitivityAnalysisResult.get_sensitivity_matrix	Get the matrix of sensitivity values on the base case or on post contingency state.
SensitivityAnalysisResult.get_reference_matrix	The reference values on the base case or on post contingency state.

pypowsybl.sensitivity.DcSensitivityAnalysisResult

class DcSensitivityAnalysisResult(*result_context_ptr*, *functions_ids*, *function_data_frame_index*)

Represents the result of a DC sensitivity analysis.

The result contains computed values (so called "reference" values) and sensitivity values of requested factors, on the base case and on post contingency states.

Methods

__init__(result_context_ptr, functions_ids, ...)	
clean_contingency_id(contingency_id)	
get_branch_flows_sensitivity_matrix([...])	
get_reference_flows([matrix_id, contingency_id])	
get_reference_matrix([matrix_id, ...])	The reference values on the base case or on post contingency state.
get_sensitivity_matrix([matrix_id, ...])	Get the matrix of sensitivity values on the base case or on post contingency state.
process_ptdf(df, matrix_id)	

Parameters

- **result_context_ptr** (*pypowsybl._pypowsybl.JavaHandle*)
- **functions_ids** (*Dict[str, List[str]]*)
- **function_data_frame_index** (*Dict[str, List[str]]*)

pypowsybl.sensitivity.DcSensitivityAnalysisResult.get_branch_flows_sensitivity_matrix

DcSensitivityAnalysisResult.**get_branch_flows_sensitivity_matrix**(*matrix_id='default'*, *contingency_id=None*)

Deprecated since version 1.1.0: Use `get_sensitivity_matrix()` instead.

Get the matrix of branch flows sensitivities on the base case or on post contingency state.

If contingency_id is None, returns the base case matrix.

Parameters

- **matrix_id** (*str*) – ID of the matrix
- **contingency_id** (*str | None*) – ID of the contingency

Returns
the matrix of branch flows sensitivities

Return type
DataFrame | None

pypowsybl.sensitivity.DcSensitivityAnalysisResult.get_reference_flows

DcSensitivityAnalysisResult.**get_reference_flows**(*matrix_id='default'*, *contingency_id=None*)

Deprecated since version 1.1.0: Use `get_reference_matrix()` instead.

The branches active power flows on the base case or on post contingency state.

Parameters

- **matrix_id** (*str*) – ID of the matrix
- **contingency_id** (*str | None*) – ID of the contingency

Returns
the branches active power flows

Return type
DataFrame | None

pypowsybl.sensitivity.AcSensitivityAnalysisResult

class AcSensitivityAnalysisResult(*result_context_ptr*, *functions_ids*, *function_data_frame_index*)

Represents the result of an AC sensitivity analysis.

The result contains computed values (so-called "reference" values) and sensitivity values of requested factors, on the base case and on post contingency states.

Methods

__init__(result_context_ptr, functions_ids, ...)	
clean_contingency_id(contingency_id)	
get_branch_flows_sensitivity_matrix([...])	
get_bus_voltages_sensitivity_matrix([...])	
get_reference_flows([matrix_id, contingency_id])	
get_reference_matrix([matrix_id, ...])	The reference values on the base case or on post contingency state.
get_reference_voltages([matrix_id, ...])	
get_sensitivity_matrix([matrix_id, ...])	Get the matrix of sensitivity values on the base case or on post contingency state.
process_ptdf(df, matrix_id)	

Parameters

- **result_context_ptr** (*pypowsybl._pypowsybl.JavaHandle*)
- **functions_ids** (*Dict[str, List[str]]*)
- **function_data_frame_index** (*Dict[str, List[str]]*)

pypowsybl.sensitivity.AcSensitivityAnalysisResult.get_bus_voltages_sensitivity_matrix

AcSensitivityAnalysisResult.**get_bus_voltages_sensitivity_matrix**(*matrix_id='default'*, *contingency_id=None*)

Deprecated since version 1.1.0: Use `get_sensitivity_matrix()` instead.

Get the matrix of bus voltages sensitivities on the base case or on post contingency state.

Parameters

- **contingency_id** (*str | None*) – ID of the contingency
- **matrix_id** (*str*)

Returns
the matrix of sensitivities

Return type
DataFrame | None

pypowsybl.sensitivity.AcSensitivityAnalysisResult.get_reference_voltages

`AcSensitivityAnalysisResult.get_reference_voltages(matrix_id='default', contingency_id=None)`

Deprecated since version 1.1.0: Use `get_reference_matrix()` instead.

The values of bus voltages on the base case or on post contingency state.

Parameters

- **matrix_id** (*str*) – ID of the matrix
- **contingency_id** (*str | None*) – ID of the contingency

Returns
the values of bus voltages

Return type
DataFrame | None

pypowsybl.sensitivity.SensitivityAnalysisResult

`class SensitivityAnalysisResult(result_context_ptr, functions_ids, function_data_frame_index)`

Represents the result of a sensitivity analysis.

The result contains computed values (so called "reference" values) and sensitivity values of requested factors, on the base case and on post contingency states.

Methods

`__init__`(result_context_ptr, functions_ids, ...)	
`clean_contingency_id`(contingency_id)	
get_reference_matrix([matrix_id, ...])	The reference values on the base case or on post contingency state.
get_sensitivity_matrix([matrix_id, ...])	Get the matrix of sensitivity values on the base case or on post contingency state.
`process_ptdf`(df, matrix_id)	

Parameters

- **result_context_ptr** (*pypowsybl._pypowsybl.JavaHandle*)
- **functions_ids** (*Dict[str, List[str]]*)
- **function_data_frame_index** (*Dict[str, List[str]]*)

pypowsybl.sensitivity.SensitivityAnalysisResult.get_sensitivity_matrix

`SensitivityAnalysisResult.get_sensitivity_matrix(matrix_id='default', contingency_id=None)`

Get the matrix of sensitivity values on the base case or on post contingency state.

If contingency_id is None, returns the base case matrix.

Parameters

- **matrix_id** (*str*) – ID of the matrix

- **contingency_id** (*str | None*) – ID of the contingency

Returns
the matrix of sensitivity values

Return type
DataFrame | None

pypowsybl.sensitivity.SensitivityAnalysisResult.get_reference_matrix

SensitivityAnalysisResult.**get_reference_matrix**(*matrix_id='default'*, *contingency_id=None*, *reference_column_id='reference_values'*)

The reference values on the base case or on post contingency state.

Parameters

- **matrix_id** (*str*) – ID of the matrix
- **contingency_id** (*str | None*) – ID of the contingency
- **reference_column_id** (*str*)

Returns
the reference values

Return type
DataFrame | None

GLSK UCTE file loading

UCTE GLSK files can be loaded using glsk.load and GLSKDocument, data can be used for zone creation.

load	Loads a GLSK file.
GLSKDocument	Result of GLSK file parsing, provides access to underlying data.
GLSKDocument.get_gsk_time_interval_start	
GLSKDocument.get_gsk_time_interval_end	
GLSKDocument.get_countries	
GLSKDocument.get_points_for_country	
GLSKDocument.get_glsk_factors	

pypowsybl.glsk.load

load(*file*)

Loads a GLSK file.

Parameters
file (*str | PathLike*) – path to the GLSK file

Returns
A GLSK document object.

Return type
GLSKDocument

pypowsybl.glsk.GLSKDocument

class GLSKDocument(*handle*)

Result of GLSK file parsing, provides access to underlying data.

Methods

__init__(handle)	
get_countries()	
get_glsk_factors(network, country, instant)	
get_gsk_time_interval_end()	
get_gsk_time_interval_start()	
get_points_for_country(network, country, instant)	

Parameters

handle (*pypowsybl._pypowsybl.JavaHandle*)

pypowsybl.glsk.GLSKDocument.get_gsk_time_interval_start

GLSKDocument.**get_gsk_time_interval_start**()

Return type

datetime

pypowsybl.glsk.GLSKDocument.get_gsk_time_interval_end

GLSKDocument.**get_gsk_time_interval_end**()

Return type

datetime

pypowsybl.glsk.GLSKDocument.get_countries

GLSKDocument.**get_countries**()

Return type

List[str]

pypowsybl.glsk.GLSKDocument.get_points_for_country

GLSKDocument.**get_points_for_country**(*network*, *country*, *instant*)

Parameters

- **network** (Network)
- **country** (*str*)
- **instant** (*datetime*)

Return type

List[str]

pypowsybl.glsk.GLSKDocument.get_glsk_factors

GLSKDocument.**get_glsk_factors**(*network*, *country*, *instant*)

Parameters

- **network** (Network)
- **country** (*str*)
- **instant** (*datetime*)

Return type
List[float]

3.1.6 Flow decomposition

The flow decomposition module allows to decompose active flows on cross-border relevant network element with contingency (XNEC) based on the ACER methodology. This python interface is based on the java implementation in the PowSyBl ENTSO-E repository.

This simple version of flow decomposition will evolve with the next versions of flow decomposition Java version. Here are the assumptions that we made:

- XNEC = lines specified by the user
- zone = country
- country GSK
- no HVDC management

Run a flow decomposition

The general idea of this API is to create a decomposition object. Then, you can define contingencies if necessary. Then, you can define XNE and XNEC. XNEC definition requires pre-defined contingencies. Some pre-defined XNE selection adder functions are available. All the adder functions will be united when running a flow decomposition. Finally, you can run the flow decomposition with some flow decomposition and/or load flow parameters.

To do so, you can use the following methods:

create_decomposition	Creates a flow decomposition objet, which can be used to run a flow decomposition on a network
FlowDecomposition.add_single_element_contingencies	Add a contingency for each element (n N-1 states).
FlowDecomposition.add_multiple_elements_contingency	Add a contingency with multiple elements (1 N-k state).
FlowDecomposition.add_monitored_elements	Add branches to be monitored by the flow decomposition.
FlowDecomposition.add_precontingency_monitored_elements	Add branches to be monitored by the flow decomposition on precontingency state (XNE(s)).
FlowDecomposition.add_postcontingency_monitored_elements	Add branches to be monitored by the flow decomposition on postcontingency state (XNEC(s)).
FlowDecomposition.add_5perc_ptdf_as_monitored_elements	Add branches that have a zone to zone PTDF greater than 5% or that are interconnections to be monitored on a precontingency state (XNE).
FlowDecomposition.add_interconnections_as_monitored_elements	Add branches that are interconnections to be monitored on a precontingency state (XNE).

continues on next page

Table 70 – continued from previous page

FlowDecomposition.add_all_branches_as_monitored_elements	Add all branches of the network to be monitored on a precontingency state (XNE).
FlowDecomposition.run	Runs a flow decomposition.

pypowsybl.flowdecomposition.create_decomposition

create_decomposition()

Creates a flow decomposition objet, which can be used to run a flow decomposition on a network

Example

```
>>> flowdecomposition = pp.flowdecomposition.create_decomposition()
>>> flowdecomposition.add_monitored_elements(['line_1', 'line_2'])
>>> flowdecomposition.run(network)
```

Returns
A flow decomposition object, which allows to run a flow decomposition on a network.

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.add_single_element_contingencies

FlowDecomposition.**add_single_element_contingencies**(*element_ids*, *contingency_id_provider=None*)

Add a contingency for each element (n N-1 states).

Parameters

- **element_ids** (*List[str]*) – List of elements
- **contingency_id_provider** (*Callable[[str], str] | None*) – Function to transform an element id to a contingency id. By default, the identity function is used.

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.add_multiple_elements_contingency

FlowDecomposition.**add_multiple_elements_contingency**(*elements_ids*, *contingency_id=None*)

Add a contingency with multiple elements (1 N-k state).

Parameters

- **element_ids** – List of elements
- **contingency_id** (*str | None*) – Id of the contingency. By default, the concatenation of each element id is used.
- **elements_ids** (*List[str]*)

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.add_monitored_elements

FlowDecomposition.**add_monitored_elements**(*branch_ids*, *contingency_ids=None*, *contingency_context_type=pypowsybl._pypowsybl.ContingencyContextType.ALL*)

Add branches to be monitored by the flow decomposition. This will create a XNEC for each valid pair of branch and state. You can select the type of states you want. If you provide contingency ids, do not forget to create them **before** calling this function. If no contingency ids are provided, elements added by this function will be XNE only.

Parameters

- **branch_ids** (*List[str] | str*) – List of branches to monitor
- **contingency_ids** (*List[str] | str | None*) – List of contingencies
- **contingency_context_type** (*pypowsybl._pypowsybl.ContingencyContextType*) – Defines if the branches should be monitored for all states (ALL by default), only N situation (NONE) or only specified contingencies (SPECIFIC)

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.add_precontingency_monitored_elements

FlowDecomposition.**add_precontingency_monitored_elements**(*branch_ids*)

Add branches to be monitored by the flow decomposition on precontingency state (XNE(s)).

Parameters
branch_ids (*List[str] | str*) – List of branches to be monitored

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.add_postcontingency_monitored_elements

FlowDecomposition.**add_postcontingency_monitored_elements**(*branch_ids*, *contingency_ids*)

Add branches to be monitored by the flow decomposition on postcontingency state (XNEC(s)). This will create a XNEC for each valid pair of branch and contingency. A pair is valid if the contingency does not contain the branch. Do not forget to create contingencies **before** calling this function.

Parameters

- **branch_ids** (*List[str] | str*) – List of branches to be monitored
- **contingency_ids** (*List[str] | str*) – List of contingencies

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.add_5perc_ptdf_as_monitored_elements

FlowDecomposition.**add_5perc_ptdf_as_monitored_elements**()

Add branches that have a zone to zone PTDF greater than 5% or that are interconnections to be monitored on a precontingency state (XNE).

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.add_interconnections_as_monitored_elements

FlowDecomposition.**add_interconnections_as_monitored_elements**()

Add branches that are interconnections to be monitored on a precontingency state (XNE).

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.add_all_branches_as_monitored_elements

FlowDecomposition.**add_all_branches_as_monitored_elements**()

Add all branches of the network to be monitored on a precontingency state (XNE).

Return type
FlowDecomposition

pypowsybl.flowdecomposition.FlowDecomposition.run

FlowDecomposition.**run**(*network*, *flow_decomposition_parameters=None*, *load_flow_parameters=None*)

Runs a flow decomposition.

Parameters

- **network** (Network) – Network on which the flow decomposition will be computed
- **flow_decomposition_parameters** (Parameters | *None*) – Flow decomposition parameters
- **load_flow_parameters** (Parameters | *None*) – Load flow parameters

Returns
A dataframe with decomposed flow for each relevant line

Return type
DataFrame

Notes

The resulting dataframe, depending on the number of countries, will include the following columns:

- **branch_id**: the id of the branch
- **contingency_id**: the id of the contingency
- **country1**: the country id of terminal 1
- **country2**: the country id of terminal 2
- **ac_reference_flow**: the ac reference flow on the line (in MW)
- **dc_reference_flow**: the dc reference flow on the line (in MW)
- **commercial_flow**: the commercial (or allocated) flow on the line (in MW)
- **x_node_flow**: the flow created by unmerged xnodes (in MW)
- **pst_flow**: the PST flow on the line (in MW)
- **internal_flow**: the internal flow on the line (in MW)
- **loop_flow_from_XX**: the loop flow from zone XX on the line (in MW). One column per country

This dataframe is indexed on the xnec ID **xnec_id**.

Example

```
>>> network = pp.network.create_eurostag_tutorial_example1_network()
>>> flow_decomposition_parameters = pp.flowdecomposition.Parameters()
>>> load_flow_parameters = pp.loadflow.Parameters()
>>> branch_ids = ['NHV1_NHV2_1', 'NHV1_NHV2_2']
>>> flowdecomposition = pp.flowdecomposition.create_decomposition()
...     .add_single_element_contingencies(branch_ids)
...     .add_monitored_elements(branch_ids, branch_ids)
>>> flowdecomposition.run(network, flow_decomposition_parameters=flow_decomposition_
↪parameters, load_flow_parameters=load_flow_parameters)
```

It outputs something like:

/	branch	con-tin-gency	coun try1	coun try2	ac_refe	dc_refe	com-mer-cial_flc	x_noc	pst_f	in-ter-nal_fl	loop_flc	loop_flow	e from_fr
xnec_id													
NHV1_NHV	NHV1_		FR	BE	302.444	300.0	0.0	0.0	0.0	0.0	300.0	0.0	
NHV1_NHV	NHV1_	NHV1	FR	BE	610.562	600.0	0.0	0.0	0.0	0.0	600.0	0.0	
NHV1_NHV	NHV1_		FR	BE	302.444	300.0	0.0	0.0	0.0	0.0	300.0	0.0	
NHV1_NHV	NHV1_	NHV1	FR	BE	610.562	600.0	0.0	0.0	0.0	0.0	600.0	0.0	

Some enum classes are used in the computer:

ContingencyContextType	

pypowsybl.flowdecomposition.ContingencyContextType

class ContingencyContextType(**args*, ***kwargs*)

Parameters

The execution of the flowdecomposition can be customized using flowdecomposition parameters.

Parameters	Parameters for a flowdecomposition execution.

pypowsybl.flowdecomposition.Parameters

class Parameters(*enable_losses_compensation=None*, *losses_compensation_epsilon=None*, *sensitivity_epsilon=None*, *rescale_mode=None*, *dc_fallback_enabled_after_ac_divergence=None*, *sensitivity_variable_batch_size=None*)

Parameters for a flowdecomposition execution.

All parameters are first read from you configuration file, then overridden with the constructor arguments.

Parameters

- **enable_losses_compensation** (*bool* | *None*) – Enable losses compensation. Use `True` to enable AC losses compensation on the DC network.

- **losses_compensation_epsilon** (*float | None*) – Filter loads from the losses compensation. The loads with a too small absolute active power will be not be connected to the network. Use `pp.flowdecomposition.Parameters.DISABLE_LOSSES_COMPENSATION_EPSILON = -1` to disable filtering.
- **sensitivity_epsilon** (*float | None*) – Filter sensitivity values The absolute small sensitivity values will be ignored. Use `pp.flowdecomposition.Parameters.DISABLE_SENSITIVITY_EPSILON = -1` to disable filtering.
- **rescale_enabled** – Rescale the flow decomposition to the AC reference. Use``True`` to rescale flow decomposition to the AC reference.
- **dc_fallback_enabled_after_ac_divergence** (*bool | None*) – Defines the fallback bahavior after an AC divergence Use `True` to run DC loadflow if an AC loadflow diverges (default). Use `False` to throw an exception if an AC loadflow diverges.
- **sensitivity_variable_batch_size** (*int | None*) – Defines the chunk size for sensitivity analysis. This will reduce memory footprint of flow decomposition but increase computation time. If setting a too high value, a max integer error may be thrown.
- **rescale_mode** (*pypowsybl._pypowsybl.RescaleMode | None*)

3.1.7 Simulation with Dynawo

The dynamic module allows to run time domain simulation.

ModelMapping

ModelMapping()	class to map elements of a network to their respective dynamic behavior
ModelMapping.get_categories_names()	Get the dynamic model categories
ModelMapping.get_categories_information()	Get more informations about categories
ModelMapping.get_supported_models([...])	Get the supported dynamic models for a given category or for all categories if no category_name is given
ModelMapping.get_supported_models_informatic	Get more informations about the supported dynamic models for a given category or for all categories if no category_name is given
ModelMapping.add_dynamic_model(category_name)	Add a dynamic model from category_name
ModelMapping.add_base_load([df])	Add a load mapping
ModelMapping.add_load_one_transformer([df])	Add a load with one transformer mapping
ModelMapping.add_load_one_transformer_tap_cl	Add a load with one transformer and tap changer mapping
ModelMapping.add_load_two_transformers([df])	Add a load with two transformers mapping
ModelMapping.add_load_two_transformers_tap_c	Add a load with two transformers and tap changers mapping
ModelMapping.add_base_generator([df])	Add a base generator mapping
ModelMapping.add_synchronized_generator([df])	Add a synchronized generator mapping
ModelMapping.add_synchronous_generator([df])	Add a synchronous generator mapping
ModelMapping.add_signal_n_generator([df])	Add a signal N generator mapping
ModelMapping.add_wecc([df])	Add a WECC mapping
ModelMapping.add_grid_forming_converter([df])	Add a grid forming converter mapping
ModelMapping.add_inertial_grid([df])	Add an inertial grid mapping
ModelMapping.add_hvdc_p([df])	Add an HVDC P mapping
ModelMapping.add_hvdc_vsc([df])	Add an HVDC VSC mapping
ModelMapping.add_base_transformer([df])	Add a transformer mapping

continues on next page

Table 73 – continued from previous page

ModelMapping.add_base_static_var_compensato	Add a static var compensator mapping
ModelMapping.add_shunt([df])	Add a shunt compensator mapping
ModelMapping.add_base_line([df])	Add a line mapping
ModelMapping.add_base_bus([df])	Add a base bus mapping
ModelMapping.add_infinite_bus([df])	Add an infinite bus mapping
ModelMapping.add_overload_management_system(	Add a dynamic overload management system (not link to a network element)
ModelMapping.add_two_level_overload_manageme	Add a dynamic two level overload management system (not link to a network element)
ModelMapping.add_under_voltage_automation_sy	Add a dynamic under voltage automation system (not link to a network element)
ModelMapping.add_phase_shifter_i_automation_	Add a dynamic phase shifter I automation system (not link to a network element)
ModelMapping.add_phase_shifter_p_automation_	Add a dynamic phase shifter P automation system (not link to a network element)
ModelMapping.add_phase_shifter_blocking_i_au	Add a dynamic phase shifter blocking I automation system (not link to a network element)
ModelMapping.add_tap_changer_automation_syst	Add a dynamic tap changer automation system (not link to a network element)
ModelMapping.add_tap_changer_blocking_automa ...)	Add a dynamic tap changer blocking automation system (not link to a network element)

pypowsybl.dynamic.ModelMapping

class ModelMapping

class to map elements of a network to their respective dynamic behavior

Methods

__init__()	
add_base_bus([df])	Add a base bus mapping
add_base_generator([df])	Add a base generator mapping
add_base_line([df])	Add a line mapping
add_base_load([df])	Add a load mapping
add_base_static_var_compensator([df])	Add a static var compensator mapping
add_base_transformer([df])	Add a transformer mapping
add_dynamic_model(category_name[, df])	Add a dynamic model from category_name
add_grid_forming_converter([df])	Add a grid forming converter mapping
add_hvdc_p([df])	Add an HVDC P mapping
add_hvdc_vsc([df])	Add an HVDC VSC mapping
add_inertial_grid([df])	Add an inertial grid mapping
add_infinite_bus([df])	Add an infinite bus mapping
add_load_one_transformer([df])	Add a load with one transformer mapping
add_load_one_transformer_tap_changer([df])	Add a load with one transformer and tap changer mapping
add_load_two_transformers([df])	Add a load with two transformers mapping
add_load_two_transformers_tap_changers([df	Add a load with two transformers and tap changers mapping
add_overload_management_system([df])	Add a dynamic overload management system (not link to a network element)

continues on next page

Table 74 – continued from previous page

add_phase_shifter_blocking_i_automation_s	Add a dynamic phase shifter blocking I automation system (not link to a network element)
add_phase_shifter_i_automation_system([df])	Add a dynamic phase shifter I automation system (not link to a network element)
add_phase_shifter_p_automation_system([df])	Add a dynamic phase shifter P automation system (not link to a network element)
add_shunt([df])	Add a shunt compensator mapping
add_signal_n_generator([df])	Add a signal N generator mapping
add_synchronized_generator([df])	Add a synchronized generator mapping
add_synchronous_generator([df])	Add a synchronous generator mapping
add_tap_changer_automation_system([df])	Add a dynamic tap changer automation system (not link to a network element)
add_tap_changer_blocking_automation_syste ...)	Add a dynamic tap changer blocking automation system (not link to a network element)
add_two_level_overload_management_system([	Add a dynamic two level overload management system (not link to a network element)
add_under_voltage_automation_system([df])	Add a dynamic under voltage automation system (not link to a network element)
add_wecc([df])	Add a WECC mapping
get_categories_information()	Get more informations about categories
get_categories_names()	Get the dynamic model categories
get_supported_models([category_name])	Get the supported dynamic models for a given category or for all categories if no category_name is given
get_supported_models_information([category_	Get more informations about the supported dynamic models for a given category or for all categories if no category_name is given

pypowsybl.dynamic.ModelMapping.get_categories_names

ModelMapping.**get_categories_names**()

Get the dynamic model categories

Returns
list of the categories

Return type
List[str]

pypowsybl.dynamic.ModelMapping.get_categories_information

ModelMapping.**get_categories_information**()

Get more informations about categories

Returns
a dataframe with information about categories

Return type
DataFrame

pypowsybl.dynamic.ModelMapping.get_supported_models

ModelMapping.**get_supported_models**(*category_name=''*)

Get the supported dynamic models for a given category or for all categories if no category_name is given

Parameters
category_name (*str*) – dynamic model category name

Returns
list of the supported models

Return type
List[str]

pypowsybl.dynamic.ModelMapping.get_supported_models_information

ModelMapping.**get_supported_models_information**(*category_name=''*)

Get more informations about the supported dynamic models for a given category or for all categories if no category_name is given

Parameters
category_name (*str*) – dynamic model category name

Returns
a dataframe with information about the supported models

Return type
DataFrame

pypowsybl.dynamic.ModelMapping.add_dynamic_model

ModelMapping.**add_dynamic_model**(*category_name*, *df=None*, ***kwargs*)

Add a dynamic model from category_name

Args
category_name: dynamic model category df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **category_name** (*str*)
- **df** (*DataFrame | List[DataFrame | None] | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Examples

Using keyword arguments:

```
model_mapping.add_dynamic_model(category_name='Load'
                                static_id='LOAD',
                                parameter_set_id='lab',
                                model_name='LoadPQ')
```

Using dataframe:

```
df = pd.DataFrame.from_records(
                    index='static_id',
                    columns=['static_id', 'parameter_set_id', 'model_name'],
                    data=[('LOAD', 'lab', 'BUS', 'LoadPQ')])
model_mapping.add_dynamic_model(category_name='Load', df)
```

pypowsybl.dynamic.ModelMapping.add_base_load

ModelMapping.**add_base_load**(*df=None*, ***kwargs*)

Add a load mapping

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type

None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_base_load(static_id='LOAD',
                            parameter_set_id='lab',
                            model_name='LoadPQ')
```

pypowsybl.dynamic.ModelMapping.add_load_one_transformer

ModelMapping.**add_load_one_transformer**(*df=None*, ***kwargs*)

Add a load with one transformer mapping

Args

df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_load_one_transformer(static_id='LOAD',
                                       parameter_set_id='lt',
                                       model_name='LoadOneTransformer')
```

pypowsybl.dynamic.ModelMapping.add_load_one_transformer_tap_changer

ModelMapping.**add_load_one_transformer_tap_changer**(*df=None*, ***kwargs*)

Add a load with one transformer and tap changer mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_load_one_transformer_tap_changer(static_id='LOAD',
                                                   parameter_set_id='lt_tc',
                                                   model_name=
↪'LoadOneTransformerTapChanger')
```

pypowsybl.dynamic.ModelMapping.add_load_two_transformers

ModelMapping.**add_load_two_transformers**(*df=None*, ***kwargs*)

Add a load with two transformers mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_load_two_transformers(static_id='LOAD',
                                        parameter_set_id='ltt',
                                        model_name='LoadTwoTransformers')
```

pypowsybl.dynamic.ModelMapping.add_load_two_transformers_tap_changers

ModelMapping.**add_load_two_transformers_tap_changers**(*df=None*, ***kwargs*)

Add a load with two transformers and tap changers mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_load_two_transformers_tap_changers(static_id='LOAD',
                                                     parameter_set_id='ltt_tc',
                                                     model_name=
↪'LoadTwoTransformersTapChangers')
```

pypowsybl.dynamic.ModelMapping.add_base_generator

ModelMapping.**add_base_generator**(*df=None*, ***kwargs*)

Add a base generator mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_base_generator(static_id='GEN',
                                 parameter_set_id='gen',
                                 model_name='GeneratorFictitious')
```

pypowsybl.dynamic.ModelMapping.add_synchronized_generator

ModelMapping.**add_synchronized_generator**(*df=None*, ***kwargs*)

Add a synchronized generator mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_synchronized_generator(static_id='GEN',
                                         parameter_set_id='sgen',
                                         model_name='GeneratorPVFixed')
```

pypowsybl.dynamic.ModelMapping.add_synchronous_generator

ModelMapping.**add_synchronous_generator**(*df=None*, ***kwargs*)

Add a synchronous generator mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_synchronous_generator(static_id='GEN',
                                        parameter_set_id='ssgen',
                                        model_name=
↪'GeneratorSynchronousThreeWindings')
```

pypowsybl.dynamic.ModelMapping.add_signal_n_generator

ModelMapping.**add_signal_n_generator**(*df=None*, ***kwargs*)

Add a signal N generator mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_signal_n_generator(static_id='GEN',
                                     parameter_set_id='signal_n',
                                     model_name='GeneratorPVSignalN')
```

pypowsybl.dynamic.ModelMapping.add_wecc

ModelMapping.**add_wecc**(*df=None*, ***kwargs*)

Add a WECC mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_wecc(static_id='GEN',
                       parameter_set_id='wecc',
                       model_name='WT4BWeccCurrentSource')
```

pypowsybl.dynamic.ModelMapping.add_grid_forming_converter

ModelMapping.**add_grid_forming_converter**(*df=None*, ***kwargs*)

Add a grid forming converter mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_grid_forming_converter(static_id='GEN',
                                         parameter_set_id='gf',
                                         model_name=
↪'GridFormingConverterMatchingControl')
```

pypowsybl.dynamic.ModelMapping.add_inertial_grid

ModelMapping.**add_inertial_grid**(*df=None*, ***kwargs*)

Add an inertial grid mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_inertial_grid(static_id='GEN',
                                parameter_set_id='gen',
                                model_name='InertialGrid')
```

pypowsybl.dynamic.ModelMapping.add_hvdc_p

ModelMapping.**add_hvdc_p**(*df=None*, ***kwargs*)

Add an HVDC P mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_hvdc_p(static_id='HVDC_LINE',
                         parameter_set_id='hvdc_p',
                         model_name='HvdcPV')
```

pypowsybl.dynamic.ModelMapping.add_hvdc_vsc

ModelMapping.**add_hvdc_vsc**(*df=None*, ***kwargs*)

Add an HVDC VSC mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_hvdc_vsc(static_id='HVDC_LINE',
                           parameter_set_id='hvdc_vsc',
                           model_name='HvdcVSCDanglingP')
```

pypowsybl.dynamic.ModelMapping.add_base_transformer

ModelMapping.**add_base_transformer**(*df=None*, ***kwargs*)

Add a transformer mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_base_transformer(static_id='TFO',
                                   parameter_set_id='tfo',
                                   model_name='TransformerFixedRatio')
```

pypowsybl.dynamic.ModelMapping.add_base_static_var_compensator

ModelMapping.**add_base_static_var_compensator**(*df=None*, ***kwargs*)

Add a static var compensator mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_base_static_var_compensator(static_id='SVARC',
                                              parameter_set_id='svarc',
                                              model_name='StaticVarCompensatorPV')
```

pypowsybl.dynamic.ModelMapping.add_shunt

ModelMapping.**add_shunt**(*df=None*, ***kwargs*)

Add a shunt compensator mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_shunt(static_id='SHUNT',
                        parameter_set_id='sh',
                        model_name='ShuntB')
```

pypowsybl.dynamic.ModelMapping.add_base_line

ModelMapping.**add_base_line**(*df=None*, ***kwargs*)

Add a line mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
mmodel_mapping.add_base_line(static_id='LINE',
                             parameter_set_id='l',
                             model_name='Line')
```

pypowsybl.dynamic.ModelMapping.add_base_bus

ModelMapping.**add_base_bus**(*df=None*, ***kwargs*)

Add a base bus mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_base_bus(static_id='BUS',
                           parameter_set_id='bus',
                           model_name='Bus')
```

pypowsybl.dynamic.ModelMapping.add_infinite_bus

ModelMapping.**add_infinite_bus**(*df=None*, ***kwargs*)

Add an infinite bus mapping

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)

- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to map
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_infinite_bus(static_id='BUS',
                               parameter_set_id='inf_bus',
                               model_name='InfiniteBus')
```

pypowsybl.dynamic.ModelMapping.add_overload_management_system

ModelMapping.**add_overload_management_system**(*df=None*, ***kwargs*)

Add a dynamic overload management system (not link to a network element)

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **dynamic_model_id**: id of the overload management system
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **controlled_branch**: id of the branch controlled by the automation system
- **i_measurement**: id of the branch used for the current intensity measurement

- **i_measurement_side**: measured side of the i_measurement branch (ONE or TWO)
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_overload_management_system(dynamic_model_id='DM_OV',
                                             parameter_set_id='ov',
                                             controlled_branch='LINE1',
                                             i_measurement='LINE2',
                                             i_measurement_side='TWO',
                                             model_name='OverloadManagementSystem')
```

pypowsybl.dynamic.ModelMapping.add_two_level_overload_management_system

ModelMapping.**add_two_level_overload_management_system**(*df=None*, ***kwargs*)

Add a dynamic two level overload management system (not link to a network element)

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **dynamic_model_id**: id of the two level overload management system
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **controlled_branch**: id of the branch controlled by the automation system
- **i_measurement_1**: id of the first branch used for the current intensity measurement
- **i_measurement_1_side**: measured side of the i_measurement_1 branch (ONE or TWO)
- **i_measurement_2**: id of the second branch used for the current intensity measurement
- **i_measurement_2_side**: measured side of the i_measurement_2 branch (ONE or TWO)
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_two_level_overload_management_system(dynamic_model_id='DM_TOV',
                                                       parameter_set_id='tov',
                                                       controlled_branch= 'LINE1',
                                                       i_measurement_1='LINE1',
                                                       i_measurement_1_side='TWO',
                                                       i_measurement_2='LINE2',
                                                       i_measurement_2_side='ONE',
                                                       model_name=
↪'TwoLevelsOverloadManagementSystem')
```

pypowsybl.dynamic.ModelMapping.add_under_voltage_automation_system

ModelMapping.**add_under_voltage_automation_system**(*df=None*, ***kwargs*)

Add a dynamic under voltage automation system (not link to a network element)

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **dynamic_model_id**: id of the under voltage automation system
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **generator**: id of the generator controlled by the automation system
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_under_voltage_automation_system(dynamic_model_id='DM_UV',
                                                  parameter_set_id='psi',
                                                  generator='GEN',
                                                  model_name='UnderVoltage'
```

pypowsybl.dynamic.ModelMapping.add_phase_shifter_i_automation_system

ModelMapping.**add_phase_shifter_i_automation_system**(*df=None*, ***kwargs*)

Add a dynamic phase shifter I automation system (not link to a network element)

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **dynamic_model_id**: id of the phase shifter I automation system
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **transformer**: id of the transformer controlled by the automation system
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_phase_shifter_i_automation_system(dynamic_model_id='DM_PS_I',
                                                    parameter_set_id='psi',
                                                    transformer='TRA',
                                                    model_name='PhaseShifterI')
```

pypowsybl.dynamic.ModelMapping.add_phase_shifter_p_automation_system

ModelMapping.**add_phase_shifter_p_automation_system**(*df=None*, ***kwargs*)

Add a dynamic phase shifter P automation system (not link to a network element)

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **dynamic_model_id**: id of the phase shifter P automation system
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **transformer**: id of the transformer controlled by the automation system
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_phase_shifter_p_automation_system(dynamic_model_id='DM_PS_P',
                                                    parameter_set_id='ov',
                                                    transformer='TRA',
                                                    model_name='PhaseShifterP')
```

pypowsybl.dynamic.ModelMapping.add_phase_shifter_blocking_i_automation_system

ModelMapping.**add_phase_shifter_blocking_i_automation_system**(*df=None*, ***kwargs*)

Add a dynamic phase shifter blocking I automation system (not link to a network element)

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **dynamic_model_id**: id of the phase shifter blocking I automation system
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **phase_shifter_id**: id of the phase shifter I automation system controlled by the automation system
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_phase_shifter_blocking_i_automation_system(dynamic_model_id='DM_
↪PSB_I',
                                                             parameter_set_id='psb',
                                                             phase_shifter_id='PSI',
                                                             model_name=
↪'PhaseShifterBlockingI')
```

pypowsybl.dynamic.ModelMapping.add_tap_changer_automation_system

ModelMapping.**add_tap_changer_automation_system**(*df=None*, ***kwargs*)

Add a dynamic tap changer automation system (not link to a network element)

Args
df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **dynamic_model_id**: id of the tap changer automation system
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **static_id**: id of the load on which the tap changer is added
- **side**: transformer side of the tap changer (HIGH_VOLTAGE, LOW_VOLTAGE or NONE)
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Examples

Using keyword arguments:

```
model_mapping.add_tap_changer_automation_system(dynamic_model_id='DM_TC',
                                                parameter_set_id='tc',
                                                static_id='LOAD',
                                                side='HIGH_VOLTAGE',
                                                model_name='TapChangerAutomaton')
```

pypowsybl.dynamic.ModelMapping.add_tap_changer_blocking_automation_system

ModelMapping.**add_tap_changer_blocking_automation_system**(*df*, *tfo_df*, *mp1_df*, *mp2_df=None*, *mp3_df=None*, *mp5_df=None*, *mp4_df=None*)

Add a dynamic tap changer blocking automation system (not link to a network element)

Args

df: Primary attributes as a dataframe. tfo_df: Dataframe for transformer data. mpN_df: Dataframes for a measurement point data, the automation system can handle up to 5 measurement points, at least 1 measurement point is expected. For each measurement point dataframe, alternative points can be input (for example bus or busbar section) the first energized element found in the network will be used

Parameters

- **df** (*DataFrame*)
- **tfo_df** (*DataFrame*)
- **mp1_df** (*DataFrame*)
- **mp2_df** (*DataFrame | None*)
- **mp3_df** (*DataFrame | None*)
- **mp5_df** (*DataFrame | None*)
- **mp4_df** (*DataFrame | None*)

Return type

None

Notes

Valid attributes for the primary dataframes are:

- **dynamic_model_id**: id of the tap changer blocking automation system
- **parameter_set_id**: id of the parameter for this model given in the dynawo configuration
- **model_name**: name of the model used for the mapping (if none the default model will be used)

Valid attributes for the transformer dataframes are:

- **dynamic_model_id**: id of the tap changer blocking automation system
- **transformer_id**: id of a transformer controlled by the automation system

Valid attributes for the measurement point dataframes are:

- **dynamic_model_id**: id of the tap changer blocking automation system
- **measurement_point_id**: id of the bus or busbar section used for the voltage measurement

Examples

We need to provide 2 dataframes, 1 for tap changer blocking automation system basic data, and one for transformer data:

```
df = pd.DataFrame.from_records(
    index='dynamic_model_id',
    columns=['dynamic_model_id', 'parameter_set_id', 'u_measurements', 'model_name
```

(continues on next page)

(continued from previous page)

```
↪'],
    data=[('DM_TCB', 'tcb', 'BUS', 'TapChangerBlockingAutomaton')])
tfo_df = pd.DataFrame.from_records(
    index='dynamic_model_id',
    columns=['dynamic_model_id', 'transformer_id'],
    data=[('DM_TCB', 'TFO1'),
          ('DM_TCB', 'TFO2'),
          ('DM_TCB', 'TFO3')])
measurement1_df = pd.DataFrame.from_records(
    index='dynamic_model_id',
    columns=['dynamic_model_id', 'measurement_point_id'],
    data=[('DM_TCB', 'B1'),
          ('DM_TCB', 'BS1')])
measurement2_df = pd.DataFrame.from_records(
    index='dynamic_model_id',
    columns=['dynamic_model_id', 'measurement_point_id'],
    data=[('DM_TCB', 'B4')])
model_mapping.add_tap_changer_blocking_automation_system(df, tfo_df, measurement1_
↪df, measurement2_df)
```

EventMapping

EventMapping()	Class to map events
EventMapping.get_events_information()	Get more informations about events
EventMapping.add_disconnection([df])	Creates an equipment disconnection event
EventMapping.add_active_power_variation([df])	Creates an active power variation event on generator or load
EventMapping.add_reactive_power_variation([d	Creates a reactive power variation event on load and generator without dynamic model
EventMapping.add_reference_voltage_variatio	Creates a reference voltage variation event on synchronized and synchronous generator
EventMapping.add_node_fault([df])	Creates a bus node fault event
EventMapping.add_event_model(event_name[, df])	Add an event model with event_name

pypowsybl.dynamic.EventMapping

class EventMapping

Class to map events

Methods

__init__()	
add_active_power_variation([df])	Creates an active power variation event on generator or load
add_disconnection([df])	Creates an equipment disconnection event
add_event_model(event_name[, df])	Add an event model with event_name
add_node_fault([df])	Creates a bus node fault event
add_reactive_power_variation([df])	Creates a reactive power variation event on load and generator without dynamic model

continues on next page

Table 76 – continued from previous page

add_reference_voltage_variation([df])	Creates a reference voltage variation event on synchronized and synchronous generator
get_events_information()	Get more informations about events

pypowsybl.dynamic.EventMapping.get_events_information

EventMapping.**get_events_information**()

Get more informations about events

Returns
a dataframe with information about events

Return type
DataFrame

pypowsybl.dynamic.EventMapping.add_disconnection

EventMapping.**add_disconnection**(*df=None*, ***kwargs*)

Creates an equipment disconnection event

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the network element to disconnect
- **start_time**: timestep at which the event happens
- **disconnect_only**: the disconnection is made on the provided side only for branch equipment (ONE or TWO)

Examples

Using keyword arguments:

```
event_mapping.add_disconnection(static_id='LINE', start_time=3.3, disconnect_only=
↪'TWO')
```

pypowsybl.dynamic.EventMapping.add_active_power_variation

EventMapping.**add_active_power_variation**(*df=None*, ***kwargs*)

Creates an active power variation event on generator or load

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the load or generator affected by the event
- **start_time**: timestep at which the event happens
- **delta_p**: active power variation

Examples

Using keyword arguments:

```
event_mapping.add_active_power_variation(static_id='LOAD', start_time=14, delta_p=2)
```

pypowsybl.dynamic.EventMapping.add_reactive_power_variation

EventMapping.**add_reactive_power_variation**(*df=None*, ***kwargs*)

Creates a reactive power variation event on load and generator without dynamic model

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the load or generator affected by the event
- **start_time**: timestep at which the event happens

- **delta_q**: reactive power variation

Examples

Using keyword arguments:

```
event_mapping.add_reactive_power_variation(static_id='LOAD', start_time=14, delta_
↪q=2)
```

pypowsybl.dynamic.EventMapping.add_reference_voltage_variation

EventMapping.**add_reference_voltage_variation**(*df=None*, ***kwargs*)

Creates a reference voltage variation event on synchronized and synchronous generator

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the generator affected by the event
- **start_time**: timestep at which the event happens
- **delta_u**: reference voltage variation

Examples

Using keyword arguments:

```
event_mapping.add_reference_voltage_variation(static_id='GEN', start_time=14, delta_
↪u=2)
```

pypowsybl.dynamic.EventMapping.add_node_fault

EventMapping.**add_node_fault**(*df=None*, ***kwargs*)

Creates a bus node fault event

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **static_id**: id of the bus affected by the event
- **start_time**: timestep at which the event happens
- **fault_time:** delta with start_time at which the event ends
- **r_pu**: r pu variation
- **x_pu**: x pu variation

Examples

Using keyword arguments:

```
event_mapping.add_node_fault(static_id='BUS', start_time=12, fault_time=2, r_pu=0.1,
↪ x_pu=0.2)
```

pypowsybl.dynamic.EventMapping.add_event_model

EventMapping.**add_event_model**(*event_name*, *df=None*, ***kwargs*)

Add an event model with event_name

Args
event_name: event model name df: Attributes as a dataframe. kwargs: Attributes as keyword arguments.

Parameters

- **event_name** (*str*)
- **df** (*DataFrame | None*)
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Examples

Using keyword arguments:

```
model_mapping.add_event_model(event_name='Disconnect',
                              static_id='GEN',
                              start_time=3.3)
```

Using dataframe:

```
df = pd.DataFrame.from_records(
                      index='static_id',
                      columns=['static_id', 'start_time'],
                      data=[('GEN', 3.3), ('LOAD', 5.2)])
model_mapping.add_event_model(event_name='Disconnect', df)
```

OutputVariableMapping

OutputVariableMapping()	Class to map Curves and Final State Values
OutputVariableMapping.add_curves(model_id, ...)	Adds curves on a single dynamic model or network equipment without dynamic model
OutputVariableMapping.add_final_state_values(...)	Adds final state values on a single dynamic model or network equipment without dynamic model
OutputVariableMapping.add_dynamic_model_curves(...)	
OutputVariableMapping.add_standard_model_curves(...)	
OutputVariableMapping.add_dynamic_model_final_state_values(...)	
OutputVariableMapping.add_standard_model_final_state_values(...)	

pypowsybl.dynamic.OutputVariableMapping

class OutputVariableMapping

Class to map Curves and Final State Values

Methods

__init__()	
add_curves(model_id, variables)	Adds curves on a single dynamic model or network equipment without dynamic model
add_dynamic_model_curves(dynamic_model_id, ...)	
add_dynamic_model_final_state_values(...)	
add_final_state_values(model_id, variables)	Adds final state values on a single dynamic model or network equipment without dynamic model
add_standard_model_curves(static_id, variables)	
add_standard_model_final_state_values(...)	

pypowsybl.dynamic.OutputVariableMapping.add_curves

OutputVariableMapping.**add_curves**(*model_id*, *variables*)

Adds curves on a single dynamic model or network equipment without dynamic model

Parameters

- **model_id** (*str*) – dynamic model or equipment id
- **variables** (*List[str] | str*) – single element or list of variables names to record

Return type

None

pypowsybl.dynamic.OutputVariableMapping.add_final_state_values

OutputVariableMapping.**add_final_state_values**(*model_id*, *variables*)

Adds final state values on a single dynamic model or network equipment without dynamic model

Parameters

- **model_id** (*str*) – dynamic model or equipment id
- **variables** (*List[str] | str*) – single element or list of variables names to record

Return type
None

pypowsybl.dynamic.OutputVariableMapping.add_dynamic_model_curves

OutputVariableMapping.**add_dynamic_model_curves**(*dynamic_model_id*, *variables*)

Deprecated since version 1.15.0: Use *add_curves()* instead.

Parameters

- **dynamic_model_id** (*str*)
- **variables** (*List[str] | str*)

Return type
None

pypowsybl.dynamic.OutputVariableMapping.add_standard_model_curves

OutputVariableMapping.**add_standard_model_curves**(*static_id*, *variables*)

Deprecated since version 1.15.0: Use *add_curves()* instead.

Parameters

- **static_id** (*str*)
- **variables** (*List[str] | str*)

Return type
None

pypowsybl.dynamic.OutputVariableMapping.add_dynamic_model_final_state_values

OutputVariableMapping.**add_dynamic_model_final_state_values**(*dynamic_model_id*, *variables*)

Deprecated since version 1.15.0: Use *add_final_state_values()* instead.

Parameters

- **dynamic_model_id** (*str*)
- **variables** (*List[str] | str*)

Return type
None

pypowsybl.dynamic.OutputVariableMapping.add_standard_model_final_state_values

OutputVariableMapping.**add_standard_model_final_state_values**(*static_id*, *variables*)

Deprecated since version 1.15.0: Use *add_final_state_values()* instead.

Parameters

- **static_id** (*str*)
- **variables** (*List[str] | str*)

Return type
None

Parameters

Parameters([start_time, stop_time, ...])	Parameters for a dynamic simulation execution.
Simulation.get_provider_parameters()	Supported dynamic simulation specific parameters for a given provider.
Simulation.get_provider_parameters_names()	Get list of parameters for Dynawo provider.

pypowsybl.dynamic.Parameters

class **Parameters**(*start_time=None*, *stop_time=None*, *provider_parameters=None*)

Parameters for a dynamic simulation execution.

All parameters are first read from you configuration file, then overridden with the constructor arguments.

Parameters

- **start_time** (*float | None*) – instant of time at which the dynamic simulation begins, in seconds
- **stop_time** (*float | None*) – instant of time at which the dynamic simulation ends, in seconds
- **provider_parameters** (*Dict[str, str] | None*) – Define parameters linked to the dynamic simulation provider currently Dynawo is the only provider handled by pypowsybl

Methods

__init__([start_time, stop_time, ...])

pypowsybl.dynamic.Simulation.get_provider_parameters

static Simulation.**get_provider_parameters**()

Supported dynamic simulation specific parameters for a given provider.

Returns
dynamic simulation parameters dataframe

Return type
DataFrame

pypowsybl.dynamic.Simulation.get_provider_parameters_names

static Simulation.**get_provider_parameters_names**()

Get list of parameters for Dynawo provider.

Returns
the list of Dynawo's parameters

Return type
List[str]

Simulation

Simulation()	
Simulation.run(network, model_mapping[, ...])	Run the dynawo simulation

pypowsybl.dynamic.Simulation

class Simulation

Methods

__init__()	
get_provider_parameters()	Supported dynamic simulation specific parameters for a given provider.
get_provider_parameters_names()	Get list of parameters for Dynawo provider.
run(network, model_mapping[, event_mapping, ...])	Run the dynawo simulation

pypowsybl.dynamic.Simulation.run

Simulation.**run**(*network*, *model_mapping*, *event_mapping=None*, *timeseries_mapping=None*, *parameters=None*, *report_node=None*)

Run the dynawo simulation

Parameters

- **network** (Network)
- **model_mapping** (ModelMapping)
- **event_mapping** (EventMapping | *None*)
- **timeseries_mapping** (OutputVariableMapping | *None*)
- **parameters** (Parameters | *None*)
- **report_node** (*ReportNode* | *None*)

Return type
SimulationResult

Results

SimulationResult(handle)	Can only be instantiated by *run()*
SimulationResult.status()	Status of the simulation (SUCCESS or FAILURE)
SimulationResult.status_text()	Status text of the simulation (failure description or empty if success)
SimulationResult.curves()	Dataframe of the curves results, columns are the curves names and rows are timestep
SimulationResult.final_state_values()	Dataframe of the final state values results, first column is the fsv names, second one the final state values
SimulationResult.timeline()	Dataframe of the simulation timeline, first column is the event time, second one the model name and the third one the event message

pypowsybl.dynamic.SimulationResult

class SimulationResult(*handle*)

Can only be instantiated by *run()*

Methods

__init__(handle)	
curves()	Dataframe of the curves results, columns are the curves names and rows are timestep
final_state_values()	Dataframe of the final state values results, first column is the fsv names, second one the final state values
status()	Status of the simulation (SUCCESS or FAILURE)
status_text()	Status text of the simulation (failure description or empty if success)
timeline()	Dataframe of the simulation timeline, first column is the event time, second one the model name and the third one the event message

Parameters
handle (*pypowsybl._pypowsybl.JavaHandle*)

pypowsybl.dynamic.SimulationResult.status

SimulationResult.**status**()

Status of the simulation (SUCCESS or FAILURE)

Return type
pypowsybl._pypowsybl.DynamicSimulationStatus

pypowsybl.dynamic.SimulationResult.status_text

SimulationResult.**status_text**()

Status text of the simulation (failure description or empty if success)

Return type
str

pypowsybl.dynamic.SimulationResult.curves

SimulationResult.**curves**()

Dataframe of the curves results, columns are the curves names and rows are timestep

Return type
DataFrame

pypowsybl.dynamic.SimulationResult.final_state_values

SimulationResult.**final_state_values**()

Dataframe of the final state values results, first column is the fsv names, second one the final state values

Return type
DataFrame

pypowsybl.dynamic.SimulationResult.timeline

SimulationResult.**timeline**()

Dataframe of the simulation timeline, first column is the event time, second one the model name and the third one the event message

Return type
DataFrame

3.1.8 Short-circuit analysis

Run a short-circuit analysis

You can run a short-circuit analysis using the following methods:

create_analysis	Creates a short-circuit analysis object, which can be used to run a short-circuit analysis on a network
ShortCircuitAnalysis.run	Runs a short-circuit analysis.
set_default_provider	Set the default short-circuit analysis provider.
get_default_provider	Get the current default short-circuit analysis provider.
get_provider_names	Get list of supported provider names

pypowsybl.shortcircuit.create_analysis

create_analysis()

Creates a short-circuit analysis object, which can be used to run a short-circuit analysis on a network

Examples

```
>>> analysis = pypowsybl.shortcircuit.create_analysis()
>>> analysis.set_faults(id='F1', element_id='Bus1', r= 1, x= 2)
>>> res = analysis.run(network, parameters, provider_name)
```

Returns
A short-circuit analysis object.

Return type
ShortCircuitAnalysis

pypowsybl.shortcircuit.ShortCircuitAnalysis.run

ShortCircuitAnalysis.**run**(*network*, *parameters=None*, *provider=''*, *reporter=None*, *report_node=None*)

Runs a short-circuit analysis.

Parameters

- **network** (Network) – Network on which the short-circuit analysis will be computed
- **parameters** (Parameters | *None*) – short-circuit analysis parameters
- **provider** (*str*) – Name of the short-circuit analysis implementation provider to be used.
- **reporter** (*ReportNode* | *None*) – deprecated, use report_node instead
- **report_node** (*ReportNode* | *None*) – the reporter to be used to create an execution report, default is None (no report)

Returns
A short-circuit analysis result.

Return type
ShortCircuitAnalysisResult

pypowsybl.shortcircuit.set_default_provider

set_default_provider(*provider*)

Set the default short-circuit analysis provider.

Parameters
provider (*str*) – name of the default short-circuit analysis provider to set

Return type
None

pypowsybl.shortcircuit.get_default_provider

get_default_provider()

Get the current default short-circuit analysis provider.

Returns
the name of the current default short-circuit analysis provider

Return type
str

pypowsybl.shortcircuit.get_provider_names

get_provider_names()

Get list of supported provider names

Returns
the list of supported provider names

Return type
List[str]

Parameters

The execution of the short-circuit analysis can be customized using short-circuit analysis parameters.

Parameters	Parameters for a short-circuit analysis execution.

pypowsybl.shortcircuit.Parameters

class Parameters(*with_feeder_result=None*, *with_limit_violations=None*, *with_voltage_result=None*, *min_voltage_drop_proportional_threshold=None*, *study_type=None*, *provider_parameters=None*, *with_fortescue_result=None*, *initial_voltage_profile_mode=None*)

Parameters for a short-circuit analysis execution.

Please check the PowSyBl short-circuit APIs documentation, for detailed information.

Parameters

- **with_fortescue_result** (*bool | None*) – indicates whether the currents and voltages are to be given in three-phase magnitude or detailed with magnitude and angle on each phase. This parameter also applies to the feeder results and voltage results.
- **with_feeder_result** (*bool | None*) – indicates whether the contributions of each feeder to the short-circuit current at the fault node should be calculated.
- **with_limit_violations** (*bool | None*) – indicates whether limit violations should be returned after the calculation. If true, a
- **in** (*list of buses where the calculated short-circuit current is higher than the maximum admissible current (stored*)
- **ip_min** (*ip_max in the identifiableShortCircuit extension) or lower than the minimum admissible current (stored in*)
- **extension).** (*in the identifiableShortCircuit*)
- **with_voltage_result** (*bool | None*) – indicates whether the voltage profile should be calculated on every node of the network
- **min_voltage_drop_proportional_threshold** (*float | None*) – specifies a threshold for filtering the voltage results. Only nodes where the voltage drop due to the short circuit is greater than this property are retained.
- **study_type** (*pypowsybl._pypowsybl.ShortCircuitStudyType | None*) – specifies the type of short-circuit study. It can be SUB_TRANSIENT, TRANSIENT or STEADY_STATE.
- **initial_voltage_profile_mode** (*pypowsybl._pypowsybl.InitialVoltageProfileMode | None*) – specify how the computation is initialized. It can be NOMINAL, CONFIGURED or PREVIOUS_VALUE
- **provider_parameters** (*Dict[str, str] | None*)

Methods

`__init__`([with_feeder_result, ...])	

Create faults

You can define faults to be simulated with the following methods:

ShortCircuitAnalysis.set_faults	Define faults to be analysed in the short-circuit simulation.
ShortCircuitAnalysis.set_branch_fault	
ShortCircuitAnalysis.set_bus_fault	

pypowsybl.shortcircuit.ShortCircuitAnalysis.set_faults

ShortCircuitAnalysis.**set_faults**(*df=None*, ***kwargs*)

Define faults to be analysed in the short-circuit simulation.

Parameters

- **df** (*DataFrame | None*) – Attributes as a dataframe.
- **kwargs** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) – Attributes as keyword arguments.

Return type
None

Notes

The current implementation allows the simulation of three-phase bus faults, where the fault resistance and reactance, when specified, are connected to the ground in series.

Data may be provided as a dataframe or as keyword arguments. In the latter case, all arguments must have the same length.

Valid attributes are:

- **id**: the id of the fault.
- **element_id**: the id of the bus on which the fault will be simulated (bus/view topology).
- **r**: The fault resistance to ground, in Ohm (optional).
- **x**: The fault reactance to ground, in Ohm (optional).
- **proportional_location**: location of the fault on the branch as a percentage of the length of the branch, side 1 is the reference (optional, only for branch fault).
- **fault_type**: The fault type either BUS_FAULT or BRANCH_FAULT

Examples:

```
analysis = pypowsybl.shortcircuit.create_analysis()

# define a single fault as keyword arguments
analysis.set_faults(id='F1', element_id='Bus1', r= 0, x= 0)

# or, define multiple faults as keyword arguments
analysis.set_faults(id=['F1', 'F2'], element_id= [ 'Bus1', 'Bus2'], r= [0, 0], x=␣
↪[0,0])
```

(continues on next page)

(continued from previous page)

```
# or, define faults as a dataframe
analysis.set_faults(pd.DataFrame.from_records(index='id', data=[{'id': 'F1',
↪'element_id': buses.index[0], 'r': 1, 'x': 2}]))

# or, since resistance and reactance are not mandatory parameters
analysis.set_faults(pd.DataFrame.from_records(index='id', data=[{'id': 'F1',
↪'element_id': buses.index[0]}]))
```

pypowsybl.shortcircuit.ShortCircuitAnalysis.set_branch_fault

ShortCircuitAnalysis.**set_branch_fault**(*branch_id*, *element_id*, *r*, *x*, *proportional_location*)

Parameters

- **branch_id** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)
- **element_id** (*str*)
- **r** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)
- **x** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)
- **proportional_location** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

pypowsybl.shortcircuit.ShortCircuitAnalysis.set_bus_fault

ShortCircuitAnalysis.**set_bus_fault**(*bus_id*, *element_id*, *r*, *x*)

Parameters

- **bus_id** (*str*)
- **element_id** (*str*)
- **r** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)
- **x** (*Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*)

Return type
None

Results

When the short-circuit analysis is completed, you can inspect its results:

ShortCircuitAnalysisResult	The result of a short-circuit analysis.

continues on next page

Table 89 – continued from previous page

ShortCircuitAnalysisResult.fault_results	contains the results, for each fault, in a dataframe representation.
ShortCircuitAnalysisResult.feeder_results	contains the contributions of each feeder to the short-circuit current, in a dataframe representation.
ShortCircuitAnalysisResult.limit_violations	contains a list of all the violations after the fault, in a dataframe representation.
ShortCircuitAnalysisResult.voltage_bus_results	contains a list of all the short-circuit voltage bus results, in a dataframe representation.

pypowsybl.shortcircuit.ShortCircuitAnalysisResult

class ShortCircuitAnalysisResult(*handle*, *with_fortescue_result*)

The result of a short-circuit analysis.

Methods

__init__(handle, with_fortescue_result)	

Attributes

fault_results	contains the results, for each fault, in a dataframe representation.
feeder_results	contains the contributions of each feeder to the short-circuit current, in a dataframe representation.
limit_violations	contains a list of all the violations after the fault, in a dataframe representation.
voltage_bus_results	contains a list of all the short-circuit voltage bus results, in a dataframe representation.

Parameters

- **handle** (*pypowsybl._pypowsybl.JavaHandle*)
- **with_fortescue_result** (*bool*)

pypowsybl.shortcircuit.ShortCircuitAnalysisResult.fault_results

property ShortCircuitAnalysisResult.**fault_results: DataFrame**

contains the results, for each fault, in a dataframe representation. The rows are fault ids and the columns are: - status: the status of the computation, can be SUCCESS, NO_SHORT_CIRCUIT_DATA (in case the reactances of generators are missing), SOLVER_FAILURE or FAILURE - short_circuit_power: the value of the short-circuit power (in MVA) - time_constant: the duration before reaching the permanent short-circuit current - current: the current at the fault, either only the three-phase magnitude or detailed with magnitudes and angles on each phase (in A) - voltage: the voltage at the fault, either only the three-phase magnitude or detailed with magnitudes and angles on each phase (in kV)

pypowsybl.shortcircuit.ShortCircuitAnalysisResult.feeder_results

property ShortCircuitAnalysisResult.**feeder_results: DataFrame**

contains the contributions of each feeder to the short-circuit current, in a dataframe representation. The rows are the ids of the contributing feeder IDs, sorted by fault and the columns are the current, either in three-phase magnitude or detailed with magnitude and angle for each phase. The current magnitudes are in A. If the feeder is a branch or a three-winding transformer, the side to which the result applies. The dataframe should be empty if the with_feeder_result parameter is set to false.

pypowsybl.shortcircuit.ShortCircuitAnalysisResult.limit_violations

property ShortCircuitAnalysisResult.**limit_violations: DataFrame**

contains a list of all the violations after the fault, in a dataframe representation. The rows are the fault ids and the id of the equipment where the violation happens. The columns are: - subject_name: the name of the equipment where the violation occurs - limit_type: the type of limit violation, can be LOW_SHORT_CIRCUIT_CURRENT or HIGH_SHORT_CIRCUIT_CURRENT - limit_name - limit: the value of the limit that is violated (maximum or minimum admissible short-circuit current) in A - acceptable_duration - limit_reduction - value: the calculated short-circuit current that is too high or too low (in A) - side: in case of a limit on a branch, the side where the violation has been detected It should be empty when the parameter with_limit_violations is set to false

pypowsybl.shortcircuit.ShortCircuitAnalysisResult.voltage_bus_results

property ShortCircuitAnalysisResult.**voltage_bus_results: DataFrame**

contains a list of all the short-circuit voltage bus results, in a dataframe representation. The rows are for each fault the IDs of the buses sorted by voltage level ID and the columns are: - initial_voltage_magnitude: the initial voltage at the bus in kV - voltage_drop_proportional: the voltage drop in percent - voltage: the calculated voltage in kV It should be empty when the parameter with_voltage_result is set to false.

3.1.9 Voltage Initializer

The voltage_initializer module is a tool to initialize voltage of a network before a loadflow and can prevent divergence.

Run the voltage initializer

run(network, params[, debug])	Run voltage initializer on the network with the given params.

pypowsybl.voltage_initializer.run

run(*network*, *params*, *debug=False*)

Run voltage initializer on the network with the given params.

Parameters

- **network** (Network) – Network on which voltage initializer will run
- **params** (VoltageInitializerParameters) – The parameters used to customize the run
- **debug** (*bool*) – if true, the tmp directory of the voltage initializer run will not be erased.

Return type
VoltageInitializerResults

VoltageInitializerParameters : How to parameterize the tool

VoltageInitializerParameters()	Parameters of a voltage initializer run.
VoltageInitializerParameters.add_variable_shunt_compensators(...)	Indicate to voltage initializer that the given shunt compensator has a variable susceptance.
VoltageInitializerParameters.add_constant_q_generators(...)	Indicate to voltage initializer that the given generator have a constant target reactive power.
VoltageInitializerParameters.add_variable_two_windings_transformers(...)	Indicate to voltage initializer that the given 2wt have a variable ratio.
VoltageInitializerParameters.add_specific_low_voltage_limits(...)	Indicate to voltage initializer to override the network low voltages limits, limit can be given relative to former limit or absolute.
VoltageInitializerParameters.add_specific_high_voltage_limits(...)	Indicate to voltage initializer to override the network high voltages limits, limit can be given relative to previous limit or absolute.
VoltageInitializerParameters.add_specific_voltage_limits(limits)	Indicate to voltage initializer to override the network voltages limits.
VoltageInitializerParameters.set_objective(...)	If you use BETWEEN_HIGH_AND_LOW_VOLTAGE_LIMIT, you also need to call
VoltageInitializerParameters.set_objective_distance(...)	If you use BETWEEN_HIGH_AND_LOW_VOLTAGE_LIMIT, you also need to call this function.
VoltageInitializerParameters.set_log_level_ampl(...)	Changes the log level of AMPL printings.
VoltageInitializerParameters.set_log_level_solver(...)	Changes the log level of non-linear optimization solver printings.
VoltageInitializerParameters.set_reactive_slack_buses_mode(...)	Changes the log level of non-linear optimization solver printings.
VoltageInitializerParameters.set_min_plausible_low_voltage_limit(...)	Changes the minimal plausible value for low voltage limits (in p.u.) in ACOPF solving.
VoltageInitializerParameters.set_max_plausible_high_voltage_limit(...)	Changes the maximal plausible value for high voltage limits (in p.u.) in ACOPF solving.
VoltageInitializerParameters.set_active_power_variation_rate(...)	Changes the weight to favor more/less minimization of active power produced by generators.
VoltageInitializerParameters.set_min_plausible_active_power_threshold(...)	Changes the threshold of active and reactive power considered as null in the optimization.
VoltageInitializerParameters.set_low_impedance_threshold(...)	Changes the threshold of impedance considered as null.
VoltageInitializerParameters.set_min_nominal_voltage_ignored_bus(...)	Changes the threshold used to ignore voltage levels with nominal voltage lower than it.
VoltageInitializerParameters.set_min_nominal_voltage_ignored_voltage_bou	Changes the threshold used to replace voltage limits of voltage levels with nominal voltage lower than it.
VoltageInitializerParameters.set_max_plausible_power_limit(...)	Changes the threshold defining the maximum active and reactive power considered in correction of generator limits.
VoltageInitializerParameters.set_high_active_power_default_limit(...)	Changes the threshold used for the correction of high active power limit of generators.
VoltageInitializerParameters.set_low_active_power_default_limit(...)	Changes the threshold used for the correction of low active power limit of generators.
VoltageInitializerParameters.set_default_minimal_qp_range(...)	Changes the threshold used to fix active (resp.
VoltageInitializerParameters.set_default_qmax_pmax_ratio(...)	Changes the ratio used to calculate threshold for corrections of high/low reactive power limits.

continues on next page

Table 93 – continued from previous page

VoltageInitializerParameters.set_default_variable_scaling_factor(...)	Changes the scaling of all variables (except reactive slacks and transformer ratios) before ACOPF solving
VoltageInitializerParameters.set_default_constraint_scaling_factor(...)	Changes the scaling factor applied to all the constraints before ACOPF solving
VoltageInitializerParameters.set_reactive_slack_variable_scaling_factor(..	Changes the scaling factor applied to all reactive slacks variables before ACOPF solving
VoltageInitializerParameters.set_twt_ratio_variable_scaling_factor(...)	Changes the scaling factor applied to all transformer ratio variables before ACOPF solving

pypowsybl.voltage_initializer.VoltageInitializerParameters

class VoltageInitializerParameters

Parameters of a voltage initializer run.

Methods

__init__()	
add_constant_q_generators(generator_id_list)	Indicate to voltage initializer that the given generator have a constant target reactive power.
add_specific_high_voltage_limits(high_limits	Indicate to voltage initializer to override the network high voltages limits, limit can be given relative to previous limit or absolute.
add_specific_low_voltage_limits(low_limits)	Indicate to voltage initializer to override the network low voltages limits, limit can be given relative to former limit or absolute.
add_specific_voltage_limits(limits)	Indicate to voltage initializer to override the network voltages limits.
add_variable_shunt_compensators(shunt_id_list	Indicate to voltage initializer that the given shunt compensator has a variable susceptance.
add_variable_two_windings_transformers(...)	Indicate to voltage initializer that the given 2wt have a variable ratio.
set_active_power_variation_rate(...)	Changes the weight to favor more/less minimization of active power produced by generators.
set_default_constraint_scaling_factor(...)	Changes the scaling factor applied to all the constraints before ACOPF solving
set_default_minimal_qp_range(...)	Changes the threshold used to fix active (resp.
set_default_qmax_pmax_ratio(...)	Changes the ratio used to calculate threshold for corrections of high/low reactive power limits.
set_default_variable_scaling_factor(...)	Changes the scaling of all variables (except reactive slacks and transformer ratios) before ACOPF solving
set_high_active_power_default_limit(...)	Changes the threshold used for the correction of high active power limit of generators.
set_log_level_ampl(log_level_ampl)	Changes the log level of AMPL printings.
set_log_level_solver(log_level_solver)	Changes the log level of non-linear optimization solver printings.
set_low_active_power_default_limit(...)	Changes the threshold used for the correction of low active power limit of generators.
set_low_impedance_threshold(...)	Changes the threshold of impedance considered as null.

continues on next page

Table 94 – continued from previous page

set_max_plausible_high_voltage_limit(...)	Changes the maximal plausible value for high voltage limits (in p.u.) in ACOPF solving.
set_max_plausible_power_limit(...)	Changes the threshold defining the maximum active and reactive power considered in correction of generator limits.
set_min_nominal_voltage_ignored_bus(...)	Changes the threshold used to ignore voltage levels with nominal voltage lower than it.
set_min_nominal_voltage_ignored_voltage_b	Changes the threshold used to replace voltage limits of voltage levels with nominal voltage lower than it.
set_min_plausible_active_power_threshold(.	Changes the threshold of active and reactive power considered as null in the optimization.
set_min_plausible_low_voltage_limit(...)	Changes the minimal plausible value for low voltage limits (in p.u.) in ACOPF solving.
set_objective(objective)	If you use BETWEEN_HIGH_AND_LOW_VOLTAGE_LIMIT, you also need to call
set_objective_distance(distance)	If you use BETWEEN_HIGH_AND_LOW_VOLTAGE_LIMIT, you also need to call this function.
set_penalty_active_power(penalty_active_power	Sets the penalty for active power generation in the ACOPF objective.
set_penalty_invest_rea_neg(...)	
set_penalty_invest_rea_pos(...)	
set_penalty_transfo_ratio(penalty_transfo_rati	
set_penalty_units_reactive(...)	
set_penalty_voltage_target_data(...)	Sets the penalty for the voltage target data term in the ACOPF objective.
set_penalty_voltage_target_ratio(...)	Sets the penalty for the voltage target ratio term in the ACOPF objective.
set_reactive_slack_buses_mode(...)	Changes the log level of non-linear optimization solver printings.
set_reactive_slack_variable_scaling_facto	Changes the scaling factor applied to all reactive slacks variables before ACOPF solving
set_twt_ratio_variable_scaling_factor(...)	Changes the scaling factor applied to all transformer ratio variables before ACOPF solving

pypowsybl.voltage_initializer.VoltageInitializerParameters.add_variable_shunt_compensators

VoltageInitializerParameters.**add_variable_shunt_compensators**(*shunt_id_list*)

Indicate to voltage initializer that the given shunt compensator has a variable susceptance.

Parameters
shunt_id_list (*List[str]*) – List of shunt ids.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.add_constant_q_generators

VoltageInitializerParameters.**add_constant_q_generators**(*generator_id_list*)

Indicate to voltage initializer that the given generator have a constant target reactive power.

Parameters
generator_id_list (*List[str]*) – List of generator ids.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.add_variable_two_windings_transformers

VoltageInitializerParameters.**add_variable_two_windings_transformers**(*transformer_id_list*)

Indicate to voltage initializer that the given 2wt have a variable ratio.

Parameters
transformer_id_list (*List[str]*) – List of transformer ids.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.add_specific_low_voltage_limits

VoltageInitializerParameters.**add_specific_low_voltage_limits**(*low_limits*)

Indicate to voltage initializer to override the network low voltages limits, limit can be given relative to former limit or absolute. High limits can be given for the same voltage level ids using *add_specific_high_voltage_limits()* but it is not necessary to give a high limit as long as each voltage level has its limits defined and consistent after overrides (low limit < high limit, low limit > 0...) Use this if voltage initializer cannot converge because of infeasibility.

Parameters
low_limits (*List[Tuple[str, bool, float]]*) – A List with elements as (voltage level id, is limit relative, limit value)

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.add_specific_high_voltage_limits

VoltageInitializerParameters.**add_specific_high_voltage_limits**(*high_limits*)

Indicate to voltage initializer to override the network high voltages limits, limit can be given relative to previous limit or absolute. Low limits can be given for the same voltage level ids using *add_specific_low_voltage_limits()* but it is not necessary to give a low limit as long as each voltage level has its limits defined and consistent after overrides (low limit < high limit, low limit > 0...) Use this if voltage initializer cannot converge because of infeasibility.

Parameters
high_limits (*List[Tuple[str, bool, float]]*) – A List with elements as (voltage level id, is limit relative, limit value)

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.add_specific_voltage_limits

VoltageInitializerParameters.**add_specific_voltage_limits**(*limits*)

Indicate to voltage initializer to override the network voltages limits. Limits are given relative to previous limits. Use this if voltage initializer cannot converge because of infeasibility.

Parameters
limits (*Dict[str, Tuple[float, float]]*) – A dictionary where keys are voltage ids, values are (lower limit, upper limit)

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_objective

VoltageInitializerParameters.**set_objective**(*objective*)

If you use BETWEEN_HIGH_AND_LOW_VOLTAGE_LIMIT, you also need to call *set_objective_distance()*.

Parameters
objective (*pypowsybl._pypowsybl.VoltageInitializerObjective*) – objective function to set for VoltageInitializer.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_objective_distance

VoltageInitializerParameters.**set_objective_distance**(*distance*)

If you use BETWEEN_HIGH_AND_LOW_VOLTAGE_LIMIT, you also need to call this function.

Parameters
distance (*float*) – is in %. A 0% objective means the model will target lower voltage limit. A 100% objective means the model will target upper voltage limit.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_log_level_ampl

VoltageInitializerParameters.**set_log_level_ampl**(*log_level_ampl*)

Changes the log level of AMPL printings.

log_level_ampl can be:

- DEBUG
- INFO
- WARNING
- ERROR

Parameters
log_level_ampl (*pypowsybl._pypowsybl.VoltageInitializerLogLevelAmpl*) – the log level.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_log_level_solver

VoltageInitializerParameters.**set_log_level_solver**(*log_level_solver*)

Changes the log level of non-linear optimization solver printings.

log_level_solver can be:

- NOTHING

- ONLY_RESULTS
- EVERYTHING

Parameters
log_level_solver (*pypowsybl._pypowsybl.VoltageInitializerLogLevelSolver*) – the log level.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_reactive_slack_buses_mode

VoltageInitializerParameters.**set_reactive_slack_buses_mode**(*reactive_slack_buses_mode*)

Changes the log level of non-linear optimization solver printings.

log_level_solver can be:

- NOTHING
- ONLY_RESULTS
- EVERYTHING

Parameters

- **log_level_solver** – the log level.
- **reactive_slack_buses_mode** (*pypowsybl._pypowsybl.VoltageInitializerReactiveSlackBusesMode*)

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_min_plausible_low_voltage_limit

VoltageInitializerParameters.**set_min_plausible_low_voltage_limit**(*min_plausible_low_voltage_level*)

Changes the minimal plausible value for low voltage limits (in p.u.) in ACOPF solving.

Parameters
min_plausible_low_voltage_level (*float*) – is >= 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_max_plausible_high_voltage_limit

VoltageInitializerParameters.**set_max_plausible_high_voltage_limit**(*max_plausible_high_voltage_limit*)

Changes the maximal plausible value for high voltage limits (in p.u.) in ACOPF solving.

Parameters
max_plausible_high_voltage_limit (*float*) – is > 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_active_power_variation_rate

VoltageInitializerParameters.**set_active_power_variation_rate**(*active_power_variation_rate*)

Changes the weight to favor more/less minimization of active power produced by generators.

Parameters
active_power_variation_rate (*float*) – is >= 0 and =< 1. A 0 active_power_variation_rate means the model will minimize the sum of generations. A 1 active_power_variation_rate means the model will minimize the sum of squared differences between target and value.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_min_plausible_active_power_threshold

VoltageInitializerParameters.**set_min_plausible_active_power_threshold**(*min_plausible_active_power_threshold*)

Changes the threshold of active and reactive power considered as null in the optimization.

Parameters
min_plausible_active_power_threshold (*float*) – is >= 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_low_impedance_threshold

VoltageInitializerParameters.**set_low_impedance_threshold**(*low_impedance_threshold*)

Changes the threshold of impedance considered as null.

Parameters
low_impedance_threshold (*float*) – is >= 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_min_nominal_voltage_ignored_bus

VoltageInitializerParameters.**set_min_nominal_voltage_ignored_bus**(*min_nominal_voltage_ignored_bus*)

Changes the threshold used to ignore voltage levels with nominal voltage lower than it.

Parameters
min_nominal_voltage_ignored_bus (*float*) – is >= 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_min_nominal_voltage_ignored_voltage_bounds

VoltageInitializerParameters.**set_min_nominal_voltage_ignored_voltage_bounds**(*min_nominal_voltage_ignored_volt*

Changes the threshold used to replace voltage limits of voltage levels with nominal voltage lower than it.

Parameters
min_nominal_voltage_ignored_voltage_bounds (*float*) – is >= 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_max_plausible_power_limit

VoltageInitializerParameters.**set_max_plausible_power_limit**(*max_plausible_power_limit*)

Changes the threshold defining the maximum active and reactive power considered in correction of generator limits.

Parameters
max_plausible_power_limit (*float*) – is > 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_high_active_power_default_limit

VoltageInitializerParameters.**set_high_active_power_default_limit**(*high_active_power_default_limit*)

Changes the threshold used for the correction of high active power limit of generators.

Parameters
high_active_power_default_limit (*float*) – is > 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_low_active_power_default_limit

VoltageInitializerParameters.**set_low_active_power_default_limit**(*low_active_power_default_limit*)

Changes the threshold used for the correction of low active power limit of generators.

Parameters
low_active_power_default_limit (*float*) – is > 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_default_minimal_qp_range

VoltageInitializerParameters.**set_default_minimal_qp_range**(*default_minimal_qp_range*)

Changes the threshold used to fix active (resp. reactive) power of generators with active (resp. reactive) power limits that are closer than it.

Parameters
default_minimal_qp_range (*float*) – is >= 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_default_qmax_pmax_ratio

VoltageInitializerParameters.**set_default_qmax_pmax_ratio**(*default_qmax_pmax_ratio*)

Changes the ratio used to calculate threshold for corrections of high/low reactive power limits.

Parameters
default_qmax_pmax_ratio (*float*) – is > 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_default_variable_scaling_factor

VoltageInitializerParameters.**set_default_variable_scaling_factor**(*default_variable_scaling_factor*)

Changes the scaling of all variables (except reactive slacks and transformer ratios) before ACOPF solving

Parameters
default_variable_scaling_factor (*float*) – is > 0. Default scaling factor

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_default_constraint_scaling_factor

VoltageInitializerParameters.**set_default_constraint_scaling_factor**(*default_constraint_scaling_factor*)

Changes the scaling factor applied to all the constraints before ACOPF solving

Parameters
default_constraint_scaling_factor (*float*) – is >= 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_reactive_slack_variable_scaling_factor

VoltageInitializerParameters.**set_reactive_slack_variable_scaling_factor**(*reactive_slack_variable_scaling_factor*)

Changes the scaling factor applied to all reactive slacks variables before ACOPF solving

Parameters
reactive_slack_variable_scaling_factor (*float*) – is > 0.

Return type
None

pypowsybl.voltage_initializer.VoltageInitializerParameters.set_twt_ratio_variable_scaling_factor

VoltageInitializerParameters.**set_twt_ratio_variable_scaling_factor**(*twt_ratio_variable_scaling_factor*)

Changes the scaling factor applied to all transformer ratio variables before ACOPF solving

Parameters
twt_ratio_variable_scaling_factor (*float*) – is > 0.

Return type
None

VoltageInitializerResults : How to exploit the results

VoltageInitializerResults(result_handle)	Results of a voltage initializer run.
VoltageInitializerResults.apply_all_modifications(network)	Apply all the modifications voltage initializer found to the network.
VoltageInitializerResults.status	If the optimisation failed, it can be useful to check the indicators.
VoltageInitializerResults.indicators	Returns: The indicators as a dict of the optimisation

pypowsybl.voltage_initializer.VoltageInitializerResults

class VoltageInitializerResults(*result_handle*)

Results of a voltage initializer run.

Methods

__init__(result_handle)	
apply_all_modifications(network)	Apply all the modifications voltage initializer found to the network.

Attributes

indicators	Returns: The indicators as a dict of the optimisation
status	If the optimisation failed, it can be useful to check the indicators.

Parameters

result_handle (*pypowsybl._pypowsybl.JavaHandle*)

pypowsybl.voltage_initializer.VoltageInitializerResults.apply_all_modifications

VoltageInitializerResults.apply_all_modifications(*network*)

Apply all the modifications voltage initializer found to the network.

Parameters

network (Network) – the network on which the modifications are to be applied.

Return type

None

pypowsybl.voltage_initializer.VoltageInitializerResults.status

property VoltageInitializerResults.status: pypowsybl._pypowsybl.VoltageInitializerStatus

If the optimisation failed, it can be useful to check the indicators. :returns: The status of the optimisation

pypowsybl.voltage_initializer.VoltageInitializerResults.indicators

property VoltageInitializerResults.indicators: Dict[str, str]

Returns: The indicators as a dict of the optimisation

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

Z