PFNET Python Documentation

Release 1.2.4

Tomas Tinoco De Rubira

December 04, 2016

Welcome! This is the documentation for the Python wrapper of PFNET, last updated December 04, 2016.

What is PFNET?

PFNET is a library for modeling and analyzing electric power networks. It provides data parsers, network visualization routines, and fast and customizable constraint and objective function evaluators for modeling network optimization problems.

License

PFNET is released under the BSD 2-clause license.

Contributors

- Tomas Tinoco De Rubira

- Martin Zellner (port to Python 3)

- Adam Wigington (cross-platfrom integration, testing)

Documentation Contents

GETTING STARTED

This section describes how to get started with PFNET in Python. In particular, it covers required packages, installation, and provides a quick example showing how to use this package.

1.1 Dependencies

PFNET for Python has the following dependencies:

- Numpy (>=1.8.2): the fundamental package for scientific computing in Python.
- Scipy (>=0.13.3): a collection of mathematical algorithms and functions built on top of Numpy.
- PFNET (== 1.2.4): underlying C routines wrapped by this package (`libpfnet`).
- Graphviz (>= 2.38): graph visualization library (`libgvc`) (Optional).
- Raw parser (>=1.2): library for parsing power flow files in PSSE raw format version 32 (`libraw_parser`) (Optional).

1.2 Download

The latest version of PFNET can be downloaded from https://github.com/ttinoco/PFNET.

1.3 Installation

After building the C library `libpfnet`, the PFNET Python module can be installed using:

```
> sudo python setup.py install
```

from the `python` directory of the PFNET package.

If `libpfnet` was built without visualization capabilities, the argument `--no_graphviz` should be passed to `setup.py`. Similarly, if `libpfnet` was build without raw parsing capabilities, the argument `--no_raw_parser` should be passed to `setup.py`.

The installation can be tested using nose as follows:

```
> sudo python setup.py build_ext --inplace
> nosetests -v --exe
```

1.4 Example

As a quick example of how to use the PFNET Python module, consider the task of constructing a power network from a MATPOWER-converted power flow file and computing the average bus degree. This can be done as follows:

```
>>> import numpy as np
>>> from pfnet import Network

>>> net = Network()
>>> net.load('ieee14.mat')

>>> print np.average([b.degree for b in net.buses])
2.86
```

1.5 Documentation

Requirements to build the PFNET Python documentation:

- Sphinx (>=1.4).

To build the documentation the environment variable PFNET_DOCS must be set. The generated files will be placed in the directory PFNET_DOCS/python. To generate the files, run *make* from the python/docs directory of the PFNET package.

TWO

POWER NETWORKS

This section describes how to create and analyze power networks using PFNET.

2.1 Overview

Power networks in PFNET are represented by objects of type `Network`. These objects are initially empty and need to be loaded with data contained in specific types of files. Once the data is loaded, the network and its components can be analyzed, visualized, and used to construct network optimization problems. After a network optimization problem is solved, the network object can be updated with the solution to perform further analysis.

An important attribute of the `Network` class is `base_power`. This quantity, which has units of MVA, is useful for converting power quantities in per unit system base power to MW or MVAr.

2.2 Loading Data

Power networks can be loaded with data using the `load()` class method. This method takes as input the filename of a supported power flow file. Information about the data parsers available in PFNET and the supported file formats can be found in Section *Data Parsers*. The following simple example shows how to load data from a power flow `mat` file:

```python
>>> from pfnet import Network

>>> net = Network()
>>> print net.num_buses
0

>>> net.load('ieee14.mat')
>>> print net.num_buses
14
```

2.3 Components

Power networks have several components. These are *buses*, *branches*, *generators*, *shunt devices*, *loads*, *variable generators* (*i.e.*, non-dispatchable), and *batteries*. For obtaining an overview of the components that form a network, the class method `show_components()` can be used:

```python
>>> net.show_components()

Network Components
------------------
```

```
buses          : 14
  slack        : 1
  reg by gen   : 5
  reg by tran  : 0
  reg by shunt : 0
shunts         : 1
  fixed        : 1
  switched v   : 0
branches       : 20
  lines        : 17
  fixed trans  : 3
  phase shifters : 0
  tap changers v : 0
  tap changers Q : 0
generators     : 5
  slack        : 1
  reg          : 5
  P adjust     : 5
loads          : 11
  P adjust     : 0
vargens        : 0
batteries      : 0
```

2.3.1 Buses

Buses in a power network are objects of type `Bus`. Each bus has an `index`, a `number`, and a `name` attribute that can be used to identify this bus in a network. The `index` is associated with the location of the bus in the underlying C array of bus structures, while the `number` and `name` attributes are specified in the input data. An `index`, a `number`, or a `name` can be used to extract a specific bus from a network using the `Network` class methods `get_bus()`, `get_bus_by_number()`, and `get_bus_by_name()`, respectively:

```
>>> bus = net.get_bus(10)

>>> print bus.index == 10
True

>>> other_bus = net.get_bus_by_number(bus.number)

>>> print bus == other_bus
True
```

For convenience, a list of all the buses in the network is contained in the `buses` attribute of the `Network` class.

Buses in a network can have different properties. For example, some buses can be slack buses and others can have their voltage magnitudes regulated by generators, tap-changing transformers, or switched shunt devices. The `Bus` class provides methods for checking whether a bus has specific properties. The following example shows how to get a list of all the buses whose voltage magnitudes are regulated by generators:

```
>>> reg_buses = [b for b in net.buses if b.is_regulated_by_gen()]

>>> print len(reg_buses), net.get_num_buses_reg_by_gen()
5 5
```

A bus also has information about the devices that are connected to it or that are regulating its voltage magnitude. For example, the attributes `gens` and `reg_trans` contain a list of generators connected to the bus and a list of tap-changing transformers regulating its voltage magnitude, respectively.

2.3.2 Branches

Branches in a power network are objects of type `Branch` and are represented mathematically by the model described in Section 2.1.2 of [TTR2015]. Each branch has an `index` attribute that can be used to identify this branch in a network. The `Network` class method `get_branch()` can be used to extract a branch of a given `index`:

```
>>> branch = net.get_branch(5)

>>> print branch.index == 5
True
```

For convenience, a list of all the branches in the network is contained in the `branches` attribute of the `Network` class.

Branches in a power network can have different properties. For example, some branches can be transmission lines, fixed transformers, tap-changing transformers, or phase-shifting transformers. Tap-changing transformers in turn can control the reactive power flowing through the branch or the voltage magnitude of a bus. The `Branch` class provides methods for checking whether a branch has specific properties. The following example shows how to get a list of all the branches that are transmission lines:

```
>>> lines = [br for br in net.branches if br.is_line()]

>>> print len(lines), net.get_num_lines()
17 17
```

For branches that are transformers, the `Branch` class attributes `ratio` and `phase` correspond to the transformer's tap ratio and phase shift, respectively. These attributes correspond to the quantities a_{km} and ϕ_{km} of the branch model described in Section 2.1.2 of [TTR2015]. The quantity a_{mk} in this model is always one.

2.3.3 Generators

Generators in a power network are objects of type `Generator`. Each generator has an `index` attribute that can be used to identify this generator in a network. The `Network` class method `get_gen()` can be used to extract a generator of a given `index`:

```
>>> gen = net.get_gen(2)

>>> print gen.index == 2
True
```

For convenience, a list of all the generators in the network is contained in the `generators` attribute of the `Network` class.

Generators in a power network can have different properties. For example, some generators can be slack generators and others can provide bus voltage magnitude regulation. The `Generator` class provides methods for checking whether a generator has specific properties. The following example shows how to get a list of all the slack generators:

```
>>> slack_gens = [g for g in net.generators if g.is_slack()]

>>> print len(slack_gens), net.get_num_slack_gens()
1 1
```

The active and reactive powers that a generator injects into the bus to which it is connected are obtained from the `P` and `Q` attributes of the `Generator` class. These quantities are given in units of per unit `system base power`. The following example computes the total active power injected into the network by generators in units of MW:

```
>>> print sum([g.P for g in net.generators])*net.base_power
272.4
```

2.3.4 Shunt Devices

Shunt devices in a power network are objects of type `Shunt`. Each shunt has an `index` attribute that can be used to identify this shunt in a network. The `Network` class method `get_shunt()` can be used to extract a shunt of a given `index`:

```
>>> shunt = net.get_shunt(0)

>>> print shunt.index == 0
True
```

For convenience, a list of all the shunt devices in the network is contained in the `shunts` attribute of the `Network` class.

As with other network components, shunt devices can have different properties. Some shunt devices can be fixed while others can be switchable and configured to regulate a bus voltage magnitude.

2.3.5 Loads

Loads in a power network are objects of type `Load`. As with other components, the `index` attribute is used to identify a load in the network. A list of all the loads in the network is contained in the `loads` attribute of the `Network` class.

Similar to generators, the active and reactive powers that a load consumes from the bus to which it is connected are obtained from the `P` and `Q` attributes of the `Load` class. They are also given in units of per unit `system base power`.

2.3.6 Variable Generators

Variable generators in a power network are objects of type `VarGenerator`. They represent non-dispatchable energy sources such as wind generators or farms and photovoltaic power plants. As with other components, the `index` attribute is used to identify a variable generator in the network. In addition to the `index` attribute, a `name` attribute is also available, which can be used to extract a specific variable generator from the network using the `Network` class method `get_vargen_by_name()`. A list of all the variable generators in the network is also contained in the `var_generators` attribute of the `Network` class.

Similar to generators, the active and reactive powers produced by a variable generator are obtained from the `P` and `Q` attributes of the `VarGenerator` class in units of per unit `system base power`. This is the output of the device in the absence of uncertainty. When there is uncertainty, the output of the device is subject to variations about `P` that have a standard deviation given by the attribute `P_std`. Output limits of a variable generator are given by the `P_min`, `P_max`, `Q_min`, and `Q_max` attributes.

The output of variable generators in a network is subject to random variations that can be correlated, especially for devices that are "nearby". The method `create_vargen_P_sigma()` of the `Network` class allows constructing a covariance matrix for these variations based on a "correlation distance" `N` and a given correlation coefficient. The cross-covariance between the variation of two devices that are connected to buses that are less than `N` branches away from each other is set such that it is consistent with the given correlation coefficient.

Lastly, since many power network input files do not have variable generator information, these devices can be added to the network by using the `add_vargens()` method of the `Network` class.

2.3.7 Batteries

Batteries are objects of type `Battery`. In addition to an `index` field, these objects contain information such as energy level `E`, charging power `P`, and more.

2.4 Properties

A `Network` object has several quantities or `properties` that provide important information about the state of the network. The following table provides a description of each of these properties.

Names	Description	Units
`bus_v_max`	Maximum bus voltage magnitude	per unit
`bus_v_min`	Minimum bus voltage magnitude	per unit
`bus_v_vio`	Maximum bus voltage magnitude limit violation	per unit
`bus_P_mis`	Maximum absolute bus active power mismatch	MW
`bus_Q_mis`	Maximum absolute bus reactive power mismatch	MVAr
`gen_P_cost`	Total active power generation cost	$/hour
`gen_v_dev`	Maximum set point deviation of generator-regulated voltage	per unit
`gen_Q_vio`	Maximum generator reactive power limit violation	MVAr
`gen_P_vio`	Maximum generator active power limit violation	MW
`tran_v_vio`	Maximum band violation of transformer-regulated voltage	per unit
`tran_r_vio`	Maximum tap ratio limit violation of tap-changing transformer	unitless
`tran_p_vio`	Maximum phase shift limit violation of phase-shifting transformer	radians
`shunt_v_vio`	Maximum band violation of shunt-regulated voltage	per unit
`shunt_b_vio`	Maximum susceptance limit violation of switched shunt device	per unit
`load_P_util`	Total active power consumption utility	$/hour
`load_P_vio`	Maximum load active power limit violation	MW
`num_actions`	Number of control adjustments (greater than 2% of control range)	unitless

All of these properties are attributes of the `Network` class. If there is a change in the network, the class method `update_properties()` needs to be called in order for the network properties to reflect the change. The following example shows how to update and extract properties:

```
>>> print net.bus_v_max
1.09

>>> for bus in net.buses:
...     bus.v_mag = bus.v_mag + 0.1
...

>>> print net.bus_v_max
1.09

>>> net.update_properties()

>>> print net.bus_v_max
1.19
```

For convenience, all the network properties can be extracted at once in a dictionary using the `get_properties()` class method:

```
>>> properties = net.get_properties()

>>> print properties['bus_v_max']
1.19
```

2.5 Variables

Network quantities can be specified to be `variables`. This is useful to represent network quantities with vectors and turn the network properties described above as functions of these vectors.

To set network quantities as variables, the `Network` class method `set_flags()` is used. This method takes as arguments a *component type*, a *flag mask* for specifying which flags types to set, a `property mask` for targeting components with specific properties, and a `variable mask` for specifying which component quantities should be affected.

Property masks are component-specific. They can be combined using `logical OR` to make properties more complex. More information can be found in the following sections:

- *Bus Properties*
- *Branch Properties*
- *Generator Properties*
- *Load Properties*
- *Shunt Properties*
- *Variable Generator Properties*

Variable masks are also component-specific. They can be combined using `logical OR` to target more than one component quantity. More information can be found in the following sections:

- *Bus Variables*
- *Branch Variables*
- *Generator Variables*
- *Load Variables*
- *Shunt Variables*
- *Variable Generator Variables*

The following example shows how to set as variables all the voltage magnitudes and angles of buses regulated by generators:

```
>>> import pfnet as pf

>>> net = pf.Network()
>>> net.load('ieee14.mat')

>>> print net.num_vars
0

>>> net.set_flags(pf.OBJ_BUS,
...               pf.FLAG_VARS,
...               pf.BUS_PROP_REG_BY_GEN,
...               pf.BUS_VAR_VMAG|pf.BUS_VAR_VANG)

>>> print net.num_vars, 2*net.get_num_buses_reg_by_gen()
10 10
```

Network components have a `has_flags()` method that allows checking whether flags of a certain type associated with specific quantities are set.

Once variables have been set, the *vector* containing all the current variable values can be extracted using `get_var_values()`:

```
>>> values = net.get_var_values()

>>> print type(values)
<type 'numpy.ndarray'>

>>> print values.shape
(10,)
```

The network components that have quantities set as variables have indices that can be used to locate these quantities in the vector of all variable values:

```
>>> bus = [b for b in net.buses if b.is_reg_by_gen()][0]

>>> print bus.has_flags(pf.FLAG_VARS,pf.BUS_VAR_VMAG)
True

>>> bus.has_flags(pf.FLAG_VARS,pf.BUS_VAR_VANG)
True

>>> print bus.v_mag, net.get_var_values()[bus.index_v_mag]
1.09 1.09

>>> print bus.v_ang, net.get_var_values()[bus.index_v_ang]
-0.23 -0.23
```

A vector of variable values can be used to update the corresponding network quantities. This is done with the Network class method set_var_values():

```
>>> bus.has_flags(pf.FLAG_VARS,pf.BUS_VAR_VANG)
True

>>> values = net.get_var_values()

>>> print bus.v_mag
1.09

>>> values[bus.index_v_mag] = 1.20
>>> net.set_var_values(values)

>>> print bus.v_mag
1.20
```

As we will see in later, variables are also useful for constructing network optimization problems.

The class method get_var_values() can also be used to get upper or lower limits of the variables. To do this, a valid *variable value code* must be passed to this method.

In addition to the class method set_flags(), which allows specifying variables of components having certain properties, one can also use the Network class method set_flags_of_component() to specify variables of individual components. This is useful when the desired components cannot be targeted using a property mask. For example, the following code illustrates how to set as variables the voltage magnitudes of buses whose indices are multiples of three:

```
>>> net.clear_flags()

>>> for bus in net.buses:
...     if bus.index % 3 == 0:
...         net.set_flags_of_component(bus,pf.FLAG_VARS,pf.BUS_VAR_VMAG)
```

```
>>> print net.num_vars, len([b for b in net.buses if b.index % 3 == 0]), net.num_buses
5 5 14
```

2.6 Projections

As explained above, once the network variables have been set, a vector with the current values of the selected variables is obtained with the class method `get_var_values()`. To extract subvectors that contain values of specific variables, projection matrices can be used. These *matrices* can be obtained using the class method `get_var_projection()`, which take as arguments a *component type* and a `variable mask`, *e.g.*, *bus variable masks*. The next example sets the variables of the network to be the bus voltage magnitudes and angles of all the buses, extracts the vector of values of all variables, and then extracts two subvectors having only voltage magnitudes and only voltage angles, respectively:

```python
>>> import numpy as np
>>> import pfnet as pf

>>> net = pf.Network()
>>> net.load('ieee14.mat')

>>> net.set_flags(pf.OBJ_BUS,
...               pf.FLAG_VARS,
...               pf.BUS_PROP_ANY,
...               pf.BUS_VAR_VMAG|pf.BUS_VAR_VANG)

>>> print net.num_vars, 2*net.num_buses
28 28

>>> P1 = net.get_var_projection(pf.OBJ_BUS,pf.BUS_VAR_VMAG)
>>> P2 = net.get_var_projection(pf.OBJ_BUS,pf.BUS_VAR_VANG)

>>> print type(P1)
<class 'scipy.sparse.coo.coo_matrix'>

>>> x = net.get_var_values()
>>> v_mags = P1*x
>>> v_angs = P2*x

>>> print v_mags
[ 1.036  1.05   1.055  1.057  1.051  1.056  1.09   1.062  1.07   1.02
  1.019  1.01   1.045  1.06 ]

>>> print v_angs
[-0.27995081 -0.26459191 -0.26302112 -0.2581342  -0.26354472 -0.26075219
 -0.23317599 -0.23335052 -0.24818582 -0.15323991 -0.18029251 -0.22200588
 -0.0869174   0. ]

>>> print np.linalg.norm(x - (P1.T*v_mags+P2.T*v_angs))
0.0
```

2.7 Contingencies

PFNET provides a convenient way to specify and analyze network contingencies. A contingency is represented by an object of type `Contingency`, and is characterized by one or more `generator` or `branch` outages. The lists

of generator and branch outages of a contingency can be specified at construction, or by using the class methods
add_gen_outage() and add_branch_outage(), respectively. The following example shows how to construct a contingency:

```
>>> import pfnet as pf

>>> net = pf.Network()
>>> net.load('ieee14.mat')

>>> gen = net.get_gen(3)
>>> branch = net.get_branch(2)

>>> c1 = pf.Contingency(gens=[gen],branches=[branch])

>>> print c1.num_gen_outages, c1.num_branch_outages
1 1
```

Once a contingency has been constructed, it can be applied and later cleared. This is done using the class methods
apply() and clear(). The apply() function sets the specified generator and branches on outage and disconnects
them from the network. Voltage regulation and other controls provided by generator or transformers on outage are
lost. The clear() function undoes the changes made by the apply() function. The following example shows how
to apply and clear contingencies, and illustrates some of the side effects:

```
>>> print c1.has_gen_outage(gen), c1.has_branch_outage(branch)
True True

>>> gen_bus = gen.bus
>>> branch_bus = branch.bus_from

>>> # generator and branch connected to buses
>>> print gen in gen_bus.gens, branch in branch_bus.branches
True True

>>> c1.apply()

>>> print gen.is_on_outage(), branch.is_on_outage()
True True

>>> # generator and branch disconnected from buses
>>> print gen in gen_bus.gens, branch in branch_bus.branches
False False

>>> c1.clear()

>>> print gen.is_on_outage(), branch.is_on_outage()
False False

>>> # generator and branch connected to buses again
>>> print gen in gen_bus.gens, branch in branch_bus.branches
True True
```

2.8 Multiple Time Period

PFNET can also be used to represent and analyze power networks over multiple time periods. By default, the networks
created using Network(), as in all the examples above, are static. To consider multiple time periods, an argument
needs to be passed to the class constructor:

```
>>> net = pf.Network(5)

>>> print net.num_periods
5
```

In "multi-period" networks, certain quantities vary over time and hence are represented by vectors. Examples of such quantities are the *network properties*, generators powers, load powers, battery energy levels, bus voltages, etc. The example below shows how to set the load profile over the time periods and extract the maximum active power mismatches in the network at each time:

```
>>> for load in net.loads:
...     load.P = np.random.rand(5)

>>> print net.loads[0].P
[ 0.84  0.47  0.62  0.65  0.36]

>>> net.update_properties()

>>> print([net.bus_P_mis[t] for t in range(5)])
[81.92, 87.35, 86.71, 93.61, 89.90]
```

Lastly, for component quantities that can potentially vary over time, setting these quantities to be variables results in one variable for each time. For example, selecting the bus voltage magnitude of a bus to be variable leads to having one variable for each time period:

```
>>> bus = net.buses[3]

>>> net.set_flags_of_component(bus,pf.FLAG_VARS,pf.BUS_VAR_VMAG)

>>> print(net.num_vars)
5

>>> print bus.index_v_mag
[0 1 2 3 4]
```

DATA PARSERS

This section describes the different data parsers available in PFNET and the supported file types.

3.1 MATPOWER Case Files

MATPOWER is a MATLAB package for solving power flow and optimal power flow problems. It contains several power flow and optimal power flow cases defined in MATLAB files. These "M" files can be converted to CSV files using the script `mpc2mat.m`. These MATPOWER-converted CSV files have extension `.mat` and can be used to load power networks in PFNET.

3.2 ARTERE Case Files

PFNET can load networks from case files used by ARTERE, which is a software for performing power flow computations using the Newton-Raphson method. These files should have extension `.art`. Details about these data files can be found in the document "ARTERE: description of data files".

Currently, PFNET has limited support of these files. More specifically:

- Components with open breakers are ignored.

- For LTC-V devices, tap positions are treated as continuous and the optional fields are ignored.

- The SWITCH, TRFO, PSHIFT-P, TURLIM, SVC, LFRESV, BUSPART and BRAPART records are not supported.

- Computation control parameters are ignored.

3.2.1 Added Records

Variable generators, batteries, base power, etc.

3.3 RAW Case Files

If built with raw parsing capabilities, which requires linking PFNET with `libraw_parser`, PFNET can load power networks from files with extension `.raw`. These files are used by the software PSS ® E and are widely used by North American power system operators.

FOUR

VISUALIZATION

This section describes how to visualize power networks using PFNET. To have this capability, PFNET needs the Graphviz library `libgvc`.

4.1 Overview

To visualize a power network, a `Graph` objects needs to be created. To do this, one needs to specify the power `Network` that is to be associated with the graph:

```python
>>> import pfnet as pf

>>> net = pf.Network()
>>> net.load('ieee14.mat')

>>> g = pf.Graph(net)
```

Then, a layout must be created for graph. This can be done using the `Graph` class method `set_layout`. This method uses the sfdp algorithm of Graphviz.

The `Graph` class provides routines for coloring nodes (network buses) according to different criteria. For example, buses can be colored according to reactive power mismatches:

```python
>>> g.set_layout()

>>> g.color_nodes_by_mismatch(pf.BUS_MIS_REACTIVE)

>>> g.view()
```

FIVE
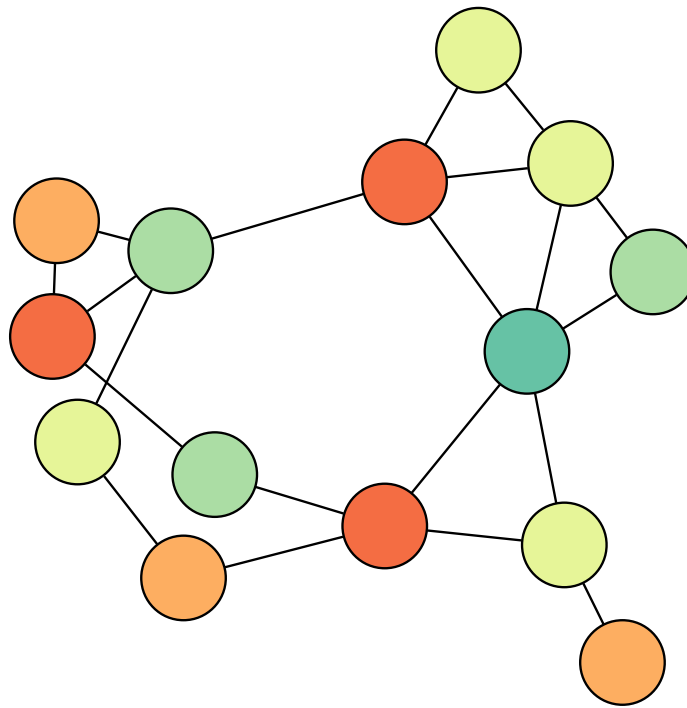
OPTIMIZATION PROBLEMS

This section describes how to formulate power network optimization problems using PFNET.

5.1 Objective Function

The objective function ϕ for a network optimization problem created using PFNET is of the form

$$\varphi(x) = \sum_i w_i \varphi_i(x),$$

where w_i are weights, φ_i are general linear or nonlinear functions, and x is a vector of values of network quantities that have been set as variables. Each weight-function pair in the summation is represented by an object of type `Function`. To instantiate an object of this type, the function type and weight need to be specified as well as the `Network` object that is to be associated with the function. The following example sets all bus voltage magnitudes as variables and constructs a function that penalizes voltage magnitude deviations from ideal values:

```
>>> import pfnet as pf

>>> net = pf.Network()
>>> net.load('ieee14.mat')

>>> net.set_flags(pf.OBJ_BUS,
...               pf.FLAG_VARS,
...               pf.BUS_PROP_ANY,
...               pf.BUS_VAR_VMAG)

>>> func = pf.Function(pf.FUNC_TYPE_REG_VMAG,0.3,net)

>>> print func.type == pf.FUNC_TYPE_REG_VMAG
True

>>> print func.weight
0.3
```

After a `Function` object is created, its value, gradient and Hessian are zero, an empty vector, and an empty matrix, respectively. Before evaluating the function at a specific vector of values, it must be analyzed using the `Function` class method `analyze()`. This routine analyzes the function and allocates the required vectors and matrices for storing its gradient and Hessian. After this, the function can be evaluated using the method `eval()`:

```
>>> x = net.get_var_values()

>>> func.analyze()

>>> func.eval(x)
```

The value $\varphi_i(x)$, gradient $\nabla \varphi_i(x)$ and Hessian $\nabla^2 \varphi_i(x)$ of a function can then be extracted from the `phi`, `gphi` and `Hphi` attributes, respectively:

```
>>> print x.shape
(14,)

>>> print func.phi
0.255

>>> print type(func.gphi), func.gphi.shape
<type 'numpy.ndarray'> (14,)

>>> print type(func.Hphi), func.Hphi.shape
<class 'scipy.sparse.coo.coo_matrix'> (14, 14)
```

For the Hessian matrix, only the lower triangular part is stored.

Details about each of the different function types available in PFNET are provided below.

5.1.1 Voltage magnitude regularization

This function has type `FUNC_TYPE_REG_VMAG`. It penalizes deviations of bus voltage magnitudes from ideal values. It is defined by the expression

$$\varphi(x) := \frac{1}{2} \sum_k \left(\frac{v_k - v_k^t}{\Delta v} \right)^2 + \frac{1}{2} \sum_k \left(\frac{v_k^y}{\Delta v} \right)^2 + \frac{1}{2} \sum_k \left(\frac{v_k^z}{\Delta v} \right)^2 + \frac{1}{2} \sum_k \left(\frac{v_k^h}{\Delta v} \right)^2 + \frac{1}{2} \sum_k \left(\frac{v_k^l}{\Delta v} \right)^2,$$

where v are bus voltage magnitudes, v^t are voltage magnitude set points (one for buses not regulated by generators), v^y and v^z are positive and negative deviations of v from v^t, v^h and v^l are voltage band upper and lower limit violations, and Δv is a normalization factor. Only terms that include optimization variables are included in the summation.

5.1.2 Voltage magnitude soft limit penalty

This function has type `FUNC_TYPE_SLIM_VMAG`. It reduces voltage (soft) limit violations by penalizing deviations of bus voltage magnitudes from the mid point of their ranges. It is defined by the expression

$$\varphi(x) := \frac{1}{2} \sum_k \left(\frac{v_k - \bar{v}_k}{\Delta v} \right)^2,$$

where v are bus voltage magnitudes, $\bar{v}$ are the mid points of their ranges, and Δv is a normalization factor. Only terms that include optimization variables are included in the summation.

5.1.3 Voltage angle regularization

This function has type `FUNC_TYPE_REG_VANG`. It penalizes large bus voltage angles and voltage angle differences across branches. It is defined by the expression

$$\varphi(x) := \frac{1}{2} \sum_k \left(\frac{\theta_k}{\Delta \theta} \right)^2 + \frac{1}{2} \sum_{(k,m)} \left(\frac{\theta_k - \theta_m - \phi_{km}}{\Delta \theta} \right)^2,$$

where θ are bus voltage angles, ϕ are branch phase shifts, and $\Delta \theta$ is a normalization factor. Only terms that include optimization variables are included in the summation.

5.1.4 Generator powers regularization

This function has type `FUNC_TYPE_REG_PQ`. It penalizes deviations of generator powers from the midpoint of their ranges. It is defined by the expression

$$\varphi(x) := \frac{1}{2} \sum_k \left(\frac{P_k^g - \bar{P}_k}{\Delta P} \right)^2 + \frac{1}{2} \sum_k \left(\frac{Q_k^g - \bar{Q}_k}{\Delta Q} \right)^2,$$

where P^g and Q^g are generator active and reactive powers, $\bar{P}$ and $\bar{Q}$ are midpoints of generator active and reactive power ranges, and $\Delta P = \Delta Q$ are normalization factors. Only terms that include optimization variables are included in the summation.

5.1.5 Active power generation cost

This function has type `FUNC_TYPE_GEN_COST`. It measures active power generation cost by the expression

$$\varphi(x) := \sum_t \sum_k q_{k0} + q_{k1} P_k(t) + q_{k2} P_k(t)^2,$$

where $P_k(t)$ are generator active powers in per unit base system power, t is the time period, and q_{k0}, q_{k1}, and q_{k2} are constant coefficients. These coefficients correspond to the attributes `cost_coeff_Q0`, `cost_coeff_Q1` and `cost_coeff_Q2` of each `Generator` object.

5.1.6 Net Active Power Consumption Cost

This function has type `FUNC_TYPE_NETCON_COST`. It measures the total cost of net active power consumption over the time periods using the price defined by the `price` attribute of each `Bus` object.

5.1.7 Active power consumption utility

This function has type `FUNC_TYPE_LOAD_UTIL`. It measures active power consumption utility by the expression

$$\varphi(x) := \sum_t \sum_k q_{k0} + q_{k1} P_k(t) + q_{k2} P_k(t)^2,$$

where $P_k(t)$ are load active powers in per unit base system power, t is the time period, and q_{k0}, q_{k1}, and q_{k2} are constant coefficients. These coefficients correspond to the attributes `util_coeff_Q0`, `util_coeff_Q1` and `util_coeff_Q2` of each `Load` object.

5.1.8 Transformer tap ratio regularization

This function has type `FUNC_TYPE_REG_RATIO`. It penalizes deviations of tap ratios of tap-changing transformers from their initial value. It is defined by the expression

$$\varphi(x) := \frac{1}{2} \sum_k \left(\frac{t_k - t_k^0}{\Delta t} \right)^2 + \frac{1}{2} \sum_k \left(\frac{t_k^y}{\Delta t} \right)^2 + \frac{1}{2} \sum_k \left(\frac{t_k^z}{\Delta t} \right)^2,$$

where t are tap ratios of tap-changing transformers, t^0 are their initial values, t^y and t^z are positive and negative deviations of t from t^0, and Δt is a normalization factor. Only terms that include optimization variables are included in the summation.

5.1.9 Transformer phase shift regularization

This function has type `FUNC_TYPE_REG_PHASE`. It penalizes deviations of phase shifts of phase shifting transformers from their initial value. It is defined by the expression

$$\varphi(x) := \frac{1}{2} \sum_k \left(\frac{\phi_k - \phi_k^0}{\Delta \phi} \right)^2$$

where ϕ are phase shifts of phase-shifting transformers, ϕ^0 are their initial values, and $\Delta \phi$ is a normalization factor. Only terms that include optimization variables are included in the summation.

5.1.10 Switched shunt susceptance regularization

This function has type `FUNC_TYPE_REG_SUSC`. It penalizes deviations of susceptances of switched shunt devices from their initial value. It is defined by the expression

$$\varphi(x) := \frac{1}{2} \sum_k \left(\frac{b_k - b_k^0}{\Delta b} \right)^2 + \frac{1}{2} \sum_k \left(\frac{b_k^y}{\Delta b} \right)^2 + \frac{1}{2} \sum_k \left(\frac{b_k^z}{\Delta b} \right)^2,$$

where b are susceptances of switched shunt devices, b^0 are their initial values, b^y and b^z are positive and negative deviations of b from b^0, and Δb is a normalization factor. Only terms that include optimization variables are included in the summation.

5.1.11 Sparsity inducing penalty for controls

This function has type `FUNC_TYPE_SP_CONTROLS`. It encourages sparse control adjustments with the expression

$$\varphi(x) := \sum_k \sqrt{\left(\frac{u_k - u_k^0}{\Delta u_k} \right)^2 + \epsilon},$$

where u are control quantities, u^0 are their current values, and ϵ is a small positive scalar. The normalization factors Δu_k are given by

$$\Delta u_k := \max\{u_k^{\max} - u_k^{\min}, \delta\},$$

where $u^{\max}$ and $u^{\min}$ are control limits, and δ is a small positive scalar. The control quantities that are considered by this function are specified using the `Network` class methods `set_flags()` or `set_flags_of_component()` using the flag type `FLAG_SPARSE`.

5.2 Constraints

Constraints in PFNET are of the form

$$Ax = b$$
$$f(x) = 0$$
$$l \leq Gx \leq u,$$

where A and G are sparse matrices, b, l and u are vectors, f is a vector-valued nonlinear function, and x is a vector of values of network quantities that have been set as variables. They are represented by objects of type `Constraint`. To create an object of this type, the constraint type and the network to be associated with the constraint need to be specified. The following example sets all bus voltage magnitudes and angles as variables and constructs the power flow constraints:

```
>>> import pfnet as pf

>>> net = pf.Network()
>>> net.load('ieee14.mat')

>>> net.set_flags(pf.OBJ_BUS,
...               pf.FLAG_VARS,
...               pf.BUS_PROP_ANY,
...               pf.BUS_VAR_VMAG|pf.BUS_VAR_VANG)

>>> print net.num_vars == 2*net.num_buses
True

>>> constr = pf.Constraint(pf.CONSTR_TYPE_PF,net)

>>> print constr.type == pf.CONSTR_TYPE_PF
True
```

Before a `Constraint` object can be used, it must be initialized using the `Constraint` class method `analyze()`. This routine analyzes the constraint and allocates the required vectors and matrices. After this, the constraint can be evaluated using the method `eval()`:

```
>>> x = net.get_var_values()

>>> constr.analyze()

>>> constr.eval(x)
```

The matrices and vectors associated with the linear constraints can be extracted from the `A`, `G`, `b`, `l` and `u` attributes of the `Constraint` object. The vector of violations and Jacobian matrix of the nonlinear constraints can be extracted from the attributes `f` and `J`, respectively. Also, the Hessian matrix of any individual nonlinear constraint $f_i(x) = 0$ can be extracted using the class method `get_H_single()`. The following example shows how to extract the largest power flow mismatch in per unit `system base power` and the Hessian matrix corresponding to the active power balance constraint of a bus:

```
>>> import numpy as np

>>> f = constr.f

>>> print type(f), f.shape
<type 'numpy.ndarray'>  (28,)

>>> print np.linalg.norm(f,np.inf)
0.042

>>> bus = net.get_bus(5)
>>> Hi = constr.get_H_single(bus.index_P)

>>> print type(Hi), Hi.shape, Hi.nnz
<class 'scipy.sparse.coo.coo_matrix'> (28, 28) 27
```

As before, all Hessian matrices have stored only the lower triangular part. In addition to being possible to extract Hessian matrices of individual nonlinear constraints, it is also possible to construct any linear combination of these individual Hessian matrices. This can be done using the `Constraint` class method `combine_H()`. After this, the resulting matrix can be extracted from the `H_combined` attribute:

```
>>> coefficients = np.random.randn(f.size)
```

```
>>> constr.combine_H(coefficients)
>>> H = constr.H_combined

>>> print type(H), H.shape, H.nnz
<class 'scipy.sparse.coo.coo_matrix'> (28, 28) 564
```

Lastly, Lagrange multiplier estimates of the linear and nonlinear constraints can be used to store sensitivity information in the network components associated with the constraints. This is done using the class method `store_sensitivities()`. Component-specific attributes that store sensitivity information are described in the *API Reference* section.

Details about each of the different constraint types available in PFNET are provided below.

5.2.1 AC Power balance

This constraint has type `CONSTR_TYPE_PF`. It enforces active and reactive power balance at every bus of the network. It is given by

$$(P_k^g + jQ_k^g) - (P_k^l + jQ_k^l) - S_k^{sh} - \sum_{m \in [n]} S_{km} = 0, \ \forall \ k \in [n],$$

where P^g and Q^g are generator active and reactive powers, P^l and Q^l are load active and reactive powers, S^{sh} are apparent powers flowing out of buses through shunt devices, S are apparent powers flowing out of buses through branches, n is the number of buses, and $[n] := \{1, \ldots, n\}$.

5.2.2 DC Power balance

This constraint has type `CONSTR_TYPE_DCPF`. It enforces "DC" active power balance at every bus of the network. It is given by

$$P_k^g - P_k^l + \sum_{m \in [n]} b_{km} (\theta_k - \theta_m - \phi_{km}) = 0, \ \forall \ k \in [n],$$

where P^g are generator active powers, P^l are load active powers, b_{km} are branch susceptances, θ_k are bus voltage angles, ϕ_{km} are phase shifts of phase-shifting transformers, n is the number of buses, and $[n] := \{1, \ldots, n\}$.

5.2.3 Linearized Power balance

This constraint has type `CONSTR_TYPE_LINPF`. It enforces active and reactive power balance at every bus of the network using a first-order Taylor expansion of the AC power balance constraints. It is given by

$$J(x_0)x = J(x_0)x_0 - f(x_0),$$

where x_0 is the vector of current variable values, $f(x_0)$ is the vector of AC bus power mismatches, and $J(x_0)$ is the Jacobian of f at x_0.

5.2.4 Branch DC power flow limits

This constraint has type `CONSTR_TYPE_DC_FLOW_LIM`. It enforces branch "DC" power flow limits due to thermal ratings. It is given by

$$-P_{km}^{\max} \leq -b_{km} (\theta_k - \theta_m - \phi_{km}) \leq P_{km}^{\max},$$

for each branch (k, m), where b_{km} are branch susceptances, θ_k are bus voltage angles, ϕ_{km} are phase shifts of phase-shifting transformers, and $P_{km}^{\max}$ are branch power flow limits.

5.2.5 Variable fixing

This constraint has type `CONSTR_TYPE_FIX`. It constrains specific variables to be fixed at their current value. The variables to be fixed are specified using the `Network` class methods `set_flags()` or `set_flags_of_component()` with the flag type `FLAG_FIXED`.

5.2.6 Variable bounding

This constraint has type `CONSTR_TYPE_BOUND`. It constrains specific variables to be inside their bounds. The variables to be bounded are specified using the `Network` class methods `set_flags()` or `set_flags_of_component()` with the flag type `FLAG_BOUNDED`. These constraints are expressed as nonlinear equality constraints using the techniques described in Section 4.3.3 of [TTR2015].

For conventional linear bounds, the constraint type `CONSTR_TYPE_LBOUND` can be used.

5.2.7 Generator participation

This constraint has type `CONSTR_TYPE_PAR_GEN`. It enforces specific active power participations among slack generators, and reactive power participations among generators regulating the same bus voltage magnitude. For slack generators, all participate with equal active powers. For voltage regulating generators, each one participates with the same fraction of its total resources. More specifically, this constraint enforces

$$P_k^g = P_m^g,$$

for all slack generators k and m connected to the same bus, and

$$\frac{Q_k^g - Q_k^{\min}}{Q_k^{\max} - Q_k^{\min}} = \frac{Q_m^g - Q_m^{\min}}{Q_m^{\max} - Q_m^{\min}},$$

for all generators k and m regulating the same bus voltage magnitude, where $Q^{\min}$ and $Q^{\max}$ are generator reactive power limits.

5.2.8 Voltage set-point regulation by generators

This constraint has type `CONSTR_TYPE_REG_GEN`. It enforces voltage set-point regulation by generators. It approximates the constraints

$$v_k = v_k^t + v_k^y - v_k^z$$
$$0 \leq (Q_k - Q_k^{\min}) \perp v_k^y \geq 0$$
$$0 \leq (Q_k^{\max} - Q_k) \perp v_k^z \geq 0,$$

for each bus k whose voltage is regulated by generators, where v are bus voltage magnitudes, v^t are their set points, v^y and v^z are positive and negative deviations of v from v^t, and Q, $Q^{\max}$ and $Q^{\min}$ are aggregate reactive powers and limits of the generators regulating the same bus voltage magnitude.

5.2.9 Voltage band regulation by transformers

This constraint has type `CONSTR_TYPE_REG_TRAN`. It enforces voltage band regulation by tap-changing transformers. It approximates the constraints

$$t_k = t_k^0 + t_k^y - t_k^z$$
$$0 \leq (v_k + v_k^l - v_k^{\min}) \perp t_k^y \geq 0$$
$$0 \leq (v_k^{\max} - v_k + v_k^h) \perp t_k^z \geq 0$$
$$0 \leq (t_k^{\max} - t_k) \perp v_k^l \geq 0$$
$$0 \leq (t_k - t_k^{\min}) \perp v_k^h \geq 0,$$

for each bus k whose voltage is regulated by tap-changing transformers, where v are bus voltage magnitudes, $v^{\max}$ and $v^{\min}$ are their band limits, v^l and v^h are voltage violations of band lower and upper limits, t are transformer tap ratios, t^0, $t^{\max}$ and $t^{\min}$ are their current values and limits, and t^y and t^z are positive and negative deviations of t from t^0. The above equations assume that the sensitivity between voltage magnitude and transformer tap ratio is positive. If it is negative, t^y and t^z are interchanged in the first two complementarity constraints, and v^l and v^h are interchanged in the bottom two complementarity constraints.

5.2.10 Voltage band regulation by switched shunts

This constraint has type `CONSTR_TYPE_REG_SHUNT`. It enforces voltage band regulation by switched shunt devices. It approximates the constraints

$$b_k = b_k^0 + b_k^y - b_k^z$$
$$0 \leq (v_k + v_k^l - v_k^{\min}) \perp b_k^y \geq 0$$
$$0 \leq (v_k^{\max} - v_k + v_k^h) \perp b_k^z \geq 0$$
$$0 \leq (b_k^{\max} - b_k) \perp v_k^l \geq 0$$
$$0 \leq (b_k - b_k^{\min}) \perp v_k^h \geq 0,$$

for each bus k whose voltage is regulated by switched shunt devices, where v are bus voltage magnitudes, $v^{\max}$ and $v^{\min}$ are their band limits, v^l and v^h are voltage violations of band lower and upper limits, b are switched shunt susceptances, b^0, $b^{\max}$ and $b^{\min}$ are their current values and limits, and b^y and b^z are positive and negative deviations of b from b^0.

5.2.11 Generator active power ramping

This constraint has type `CONSTR_TYPE_GEN_RAMP`. It enforces generator active power ramping limits. It is given by

$$-\delta P_k^{\max} \leq P_k(t) - P_k(t-1) \leq \delta P^{\max}$$

for each generator k and time period t, where $P_k(t)$ are generator active powers, and $\delta P_k^{\max}$ are generator ramping limits. The ramping limits are defined by the `dP_max` attribute of each `Generator` object. For $t = 0$, $P_k(t-1)$ is the `P_prev` attribute of a `Generator`.

5.3 Problems

Optimization problems constructed with PFNET are of the form

$$
\begin{aligned}
\text{minimize} \quad & \varphi(x) \\
\text{subject to} \quad & Ax = b \\
& f(x) = 0 \\
& l \le Gx \le u,
\end{aligned}
$$

As already noted, the objective function φ is a weighted sum of functions φ_i. The linear and nonlinear constraints $Ax = b$, $l \le Gx \le u$, and $f(x) = 0$ correspond to one or more of the constraints described above. An optimization problem in PFNET is represented by an object of type `Problem`.

After instantiation, a `Problem` is empty and one needs to specify the `Network` that is to be associated with the problem, the `Constraints` to include, and the `Functions` that form the objective function. This can be done using the `Problem` class methods `set_network()`, `add_constraint()`, and `add_function()`. The following example shows how to construct a simple power flow problem and solve it using the Newton-Raphson method:

```python
import pfnet as pf
from numpy import hstack
from numpy.linalg import norm
from scipy.sparse import bmat
from scipy.sparse.linalg import spsolve

def NRsolve(net):

    net.clear_flags()

    # bus voltage angles
    net.set_flags(pf.OBJ_BUS,
                  pf.FLAG_VARS,
                  pf.BUS_PROP_NOT_SLACK,
                  pf.BUS_VAR_VANG)

    # bus voltage magnitudes
    net.set_flags(pf.OBJ_BUS,
                  pf.FLAG_VARS,
                  pf.BUS_PROP_NOT_REG_BY_GEN,
                  pf.BUS_VAR_VMAG)

    # slack gens active powers
    net.set_flags(pf.OBJ_GEN,
                  pf.FLAG_VARS,
                  pf.GEN_PROP_SLACK,
                  pf.GEN_VAR_P)

    # regulator gens reactive powers
    net.set_flags(pf.OBJ_GEN,
                  pf.FLAG_VARS,
                  pf.GEN_PROP_REG,
                  pf.GEN_VAR_Q)

    p = pf.Problem()
    p.set_network(net)
    p.add_constraint(pf.CONSTR_TYPE_PF)          # power flow
    p.add_constraint(pf.CONSTR_TYPE_PAR_GEN_P)   # generator participation
    p.add_constraint(pf.CONSTR_TYPE_PAR_GEN_Q)   # generator participation
```

```
p.analyze()

x = p.get_init_point()
p.eval(x)

residual = lambda x: hstack((p.A*x-p.b,p.f))

while norm(residual(x)) > 1e-4:
    x = x + spsolve(bmat([[p.A],[p.J]],format='csr'),-residual(x))
    p.eval(x)

net.set_var_values(x)
net.update_properties()
```

The above routine can then be used as follows:

```
>>> net = Network()
>>> net.load('case3012wp.mat')

>>> print net.bus_P_mis, net.bus_Q_mis
2.79e+0 1.56e+1

>>> NRsolve(net)

>>> print net.bus_P_mis, net.bus_Q_mis
2.37e-6 3.58e-6
```

As shown in the example, the `Problem` class method `analyze()` needs to be called before the vectors and matrices associated with the problem constraints and functions can be used. The method `eval()` can then be used for evaluating the problem objective and constraint functions at different points. As is the case for `Constraints`, a `Problem` has a method `combine_H()` for forming linear combinations of individual constraint Hessians, and a method `store_sensitivities()` for storing sensitivity information in the network components associated with the constraints.

API REFERENCE

6.1 Vector

class numpy.**ndarray**
> See numpy documentation.

6.2 Matrix

class scipy.sparse.**coo_matrix**
> See scipy documentation.

6.3 Bus

6.3.1 Bus Properties

pfnet.**BUS_PROP_ANY**
> Any bus.

pfnet.**BUS_PROP_SLACK**
> Slack bus.

pfnet.**BUS_PROP_REG_BY_GEN**
> Bus with voltage magnitude regulated by one or more generators.

pfnet.**BUS_PROP_REG_BY_TRAN**
> Bus with voltage magnitude regulated by one or more tap-changing transformers.

pfnet.**BUS_PROP_REG_BY_SHUNT**
> Bus with voltage magnitude regulated by one or more switched shunt devices.

pfnet.**BUS_PROP_NOT_REG_BY_GEN**
> Bus with voltage magnitude that is not regulated by generators.

pfnet.**BUS_PROP_NOT_SLACK**
> Bus that is not a slack bus.

6.3.2 Bus Variables

pfnet.**BUS_VAR_VMAG**
> Bus voltage magnitude.

pfnet.**BUS_VAR_VANG**
> Bus voltage angle.

pfnet.**BUS_VAR_VDEV**
> Bus voltage magnitude positive and negative set-point deviations.

pfnet.**BUS_VAR_VVIO**
> Bus voltage magnitude upper and lower bound violations.

6.3.3 Bus Sensitivities

pfnet.**BUS_SENS_LARGEST**
> Largest objective function sensitivity with respect to constraints involving this bus.

pfnet.**BUS_SENS_P_BALANCE**
> Objective function sensitivity with respect to active power balance.

pfnet.**BUS_SENS_Q_BALANCE**
> Objective function sensitivity with respect to reactive power balance.

pfnet.**BUS_SENS_V_MAG_U_BOUND**
> Objective function sensitivity with respect to voltage magnitude upper bound.

pfnet.**BUS_SENS_V_MAG_L_BOUND**
> Objective function sensitivity with respect to voltage magnitude lower bound.

pfnet.**BUS_SENS_V_ANG_U_BOUND**
> Objective function sensitivity with respect to voltage angle upper bound.

pfnet.**BUS_SENS_V_ANG_L_BOUND**
> Objective function sensitivity with respect to voltage angle lower bound.

pfnet.**BUS_SENS_V_REG_BY_GEN**
> Objective function sensitivity with respect to voltage magnitude regulation by generators.

pfnet.**BUS_SENS_V_REG_BY_TRAN**
> Objective function sensitivity with respect to voltage magnitude regulation by tap-changing transformers.

pfnet.**BUS_SENS_V_REG_BY_SHUNT**
> Objective function sensitivity with respect to voltage magnitude regulation by switched shunt devices.

6.3.4 Bus Power Mismatches

pfnet.**BUS_MIS_LARGEST**
> Largest bus power mismatch.

pfnet.**BUS_MIS_ACTIVE**
> Bus active power mismatch.

pfnet.**BUS_MIS_REACTIVE**
> Bus reactive power mismatch.

6.3.5 Bus Class

class pfnet.**Bus**(*num_periods=1*, *alloc=True*)
> Bus class.

Parameters alloc : {`True`, `False`}

num_periods : int

P_mis

Bus active power mismatch (p.u. system base MVA) (float or array).

Q_mis

Bus reactive power mismatch (p.u. system base MVA) (float or array).

batteries

List of `batteries` connected to this bus (list).

branches

List of `branches` incident on this bus (list).

branches_from

List of `branches` that have this bus on the "from" side (list).

branches_to

List of `branches` that have this bus on the "to" side (list).

degree

Bus degree (number of incident branches) (float).

generators

List of `generators` connected to this bus (list).

get_largest_mis (*self*, *t=0*)

Gets the bus power mismatch of largest absolute value.

Parameters t : int (time period)

Returns mis : float

get_largest_mis_type (*self*, *t=0*)

Gets the type of bus power mismatch of largest absolute value.

Parameters t : int (time period)

Returns type : int

get_largest_sens (*self*, *t=0*)

Gets the bus sensitivity of largest absolute value.

Parameters t : int (time period)

Returns sens : float

get_largest_sens_type (*self*, *t=0*)

Gets the type of bus sensitivity of largest absolute value.

Parameters t : int (time period)

Returns type : int

get_num_vars (*self*, *var_mask*, *t_start=0*, *t_end=None*)

Gets number of variables associated with the given quantity.

Parameters var_mask : int

t_start : int

t_end : int

Returns num : int

get_quantity (*self*, *type*, *t=0*)
    Gets the bus quantity of the given type.

        Parameters type : int (*Bus Sensitivities*:, *Bus Power Mismatches*)

            t : int (time period)

        Returns value : float

get_total_gen_P (*self*, *t=0*)
    Gets the total active power injected by generators connected to this bus.

        Parameters t : int (time period)

        Returns P : float

get_total_gen_Q (*self*, *t=0*)
    Gets the total reactive power injected by generators connected to this bus.

        Parameters t : int (time period)

        Returns Q : float

get_total_gen_Q_max (*self*)
    Gets the largest total reactive power that can be injected by generators connected to this bus.

        Returns Q_max : float

get_total_gen_Q_min (*self*)
    Gets the smallest total reactive power that can be injected by generators connected to this bus.

        Returns Q_min : float

get_total_load_P (*self*, *t=0*)
    Gets the total active power consumed by loads connected to this bus.

        Parameters t : int (time period)

        Returns P : float

get_total_load_Q (*self*, *t=0*)
    Gets the total reactive power consumed by loads connected to this bus.

        Parameters t : int (time period)

        Returns Q : float

get_total_shunt_b (*self*, *t=0*)
    Gets the combined susceptance of shunt devices connected to this bus.

        Parameters t : int (time period)

        Returns b : float

get_total_shunt_g (*self*)
    Gets the combined conductance of shunt devices connected to this bus.

        Returns g : float

has_flags (*self*, *fmask*, *vmask*)
    Determines whether the bus has the flags associated with certain quantities set.

        Parameters fmask : int (*Flag Types*)

            vmask : int (*Bus Variables*)

        Returns flag : {True, False}

index
> Bus index (int).

index_P
> Index of bus active power mismatch (int).

index_Q
> Index for bus reactive power mismatch (int).

index_v_ang
> Index of voltage angle variable (int or array).

index_v_mag
> Index of voltage magnitude variable (int or array).

index_vh
> Index of voltage high limit violation variable (int or array).

index_vl
> Index of voltage low limit violation variable (int or array).

index_y
> Index of voltage magnitude positive deviation variable (int or array).

index_z
> Index of voltage magnitude negative deviation variable (int or array).

is_equal (*self*, *other*)
> Determines whether bus is equal to given bus.
>
> > **Parameters other** : `Bus`

is_regulated_by_gen (*self*)
> Determines whether the bus is regulated by a generator.
>
> > **Returns flag** : {`True`, `False`}

is_regulated_by_shunt (*self*)
> Determines whether the bus is regulated by a shunt device.
>
> > **Returns flag** : {`True`, `False`}

is_regulated_by_tran (*self*)
> Determines whether the bus is regulated by a transformer.
>
> > **Returns flag** : {`True`, `False`}

is_slack (*self*)
> Determines whether the bus is a slack bus.
>
> > **Returns flag** : {`True`, `False`}

loads
> List of `loads` connected to this bus (list).

name
> Bus name (sting).

num_periods
> Number of time periods (int).

number
> Bus number (int).

obj_type
Object type (int).

price
Bus energy price (float or array) ($ / (hr p.u.)).

reg_generators
List of `generators` regulating the voltage magnitude of this bus (list).

reg_shunts
List of `switched shunt devices` regulating the voltage magnitude of this bus (list).

reg_trans
List of `tap-changing transformers` regulating the voltage magnitude of this bus (list).

sens_P_balance
Objective function sensitivity with respect to bus active power balance (float or array).

sens_Q_balance
Objective function sensitivity with respect to bus reactive power balance (float or array).

sens_v_ang_l_bound
Objective function sensitivity with respect to voltage angle lower bound (float or array).

sens_v_ang_u_bound
Objective function sensitivity with respect to voltage angle upper bound (float or array).

sens_v_mag_l_bound
Objective function sensitivity with respect to voltage magnitude lower bound (float or array).

sens_v_mag_u_bound
Objective function sensitivity with respect to voltage magnitude upper bound (float or array).

sens_v_reg_by_gen
Objective function sensitivity with respect to bus voltage regulation by generators (float or array).

sens_v_reg_by_shunt
Objective function sensitivity with respect to bus voltage regulation by shunts (float or array).

sens_v_reg_by_tran
Objective function sensitivity with respect to bus voltage regulation by transformers (float or array).

set_price (*self*, *p*, *t=0*)
Sets bus energy price.

> **Parameters p** : float
>
> > **t** : int

set_v_ang (*self*, *v*, *t=0*)
Sets bus voltage angle.

> **Parameters v** : float
>
> > **t** : int

set_v_mag (*self*, *v*, *t=0*)
Sets bus voltage magnitude.

> **Parameters v** : float
>
> > **t** : int

show (*self*, *t=0*)
Shows bus properties.

> **Parameters t** : int (time period)

v_ang
> Bus voltage angle (radians) (float or array).

v_mag
> Bus volatge magnitude (p.u. bus base kv) (float or array).

v_max
> Bus volatge upper bound (p.u. bus base kv) (float).

v_min
> Bus voltage lower bound (p.u. bus base kv) (float).

v_set
> Bus voltage set point (p.u. bus base kv) (float or array). Equals one if bus is not regulated by a generator.

var_generators
> List of `variable generators` connected to this bus (list).

6.4 Branch

6.4.1 Branch Properties

`pfnet.`**`BRANCH_PROP_ANY`**
> Any branch.

`pfnet.`**`BRANCH_PROP_TAP_CHANGER`**
> Branch that is tap-changing transformer.

`pfnet.`**`BRANCH_PROP_TAP_CHANGER_V`**
> Branch that is tap-changing transformer regulating a bus voltage magnitude.

`pfnet.`**`BRANCH_PROP_TAP_CHANGER_Q`**
> Branch that is tap-changing transformer regulating reactive power flow.

`pfnet.`**`BRANCH_PROP_PHASE_SHIFTER`**
> Branch that is phase-shifting transformer regulating active power flow.

`pfnet.`**`BRANCH_PROP_NOT_OUT`**
> Branch that is not on outage.

6.4.2 Branch Variables

`pfnet.`**`BRANCH_VAR_RATIO`**
> Transformer tap ratio.

`pfnet.`**`BRANCH_VAR_RATIO_DEV`**
> Transformer tap ratio deviations from current value.

`pfnet.`**`BRANCH_VAR_PHASE`**
> Transformer phase shift.

6.4.3 Branch Class

class `pfnet.`**`Branch`**(*num_periods=1*, *alloc=True*)
> Branch class.

> **Parameters alloc** : {True, False}
>
> > **num_periods** : int

P_flow_DC
    Active power flow (DC approx.) from bus "from" to bus "to" (float or array).

b
    Branch series susceptance (p.u.) (float).

b_from
    Branch shunt susceptance at the "from" side (p.u.) (float).

b_to
    Branch shunt susceptance at the "to" side (p.u.) (float).

bus_from
    Bus connected to the "from" side.

bus_to
    Bus connected to the "to" side.

g
    Branch series conductance (p.u.) (float).

g_from
    Branch shunt conductance at the "from" side (p.u.) (float).

g_to
    Branch shunt conductance at the "to" side (p.u.) (float).

has_flags(*self*, *fmask*, *vmask*)
    Determines whether the branch has the flags associated with specific quantities set.

> **Parameters fmask** : int (*Flag Types*)
>
> > **vmask** : int (*Branch Variables*)
>
> **Returns flag** : {True, False}

has_pos_ratio_v_sens(*self*)
    Determines whether tap-changing transformer has positive sensitivity between tap ratio and controlled bus voltage magnitude.

> **Returns flag** : {True, False}

index
    Branch index (int).

index_phase
    Index of transformer phase shift variable (int or array).

index_ratio
    Index of transformer tap ratio variable (int or array).

index_ratio_y
    Index of transformer tap ratio positive deviation variable (int or array).

index_ratio_z
    Index of transformer tap ratio negative deviation variable (int or array).

is_equal(*self*, *other*)
    Determines whether branch is equal to given branch.

> **Parameters other** : Branch

is_fixed_tran(*self*)
> Determines whether branch is fixed transformer.
>
> > **Returns flag** : {`True`, `False`}

is_line(*self*)
> Determines whether branch is transmission line.
>
> > **Returns flag** : {`True`, `False`}

is_on_outage(*self*)
> Determines whether branch in on outage.
>
> > **Returns flag** : {`True`, `False`}

is_phase_shifter(*self*)
> Determines whether branch is phase shifter.
>
> > **Returns flag** : {`True`, `False`}

is_tap_changer(*self*)
> Determines whether branch is tap-changing transformer.
>
> > **Returns flag** : {`True`, `False`}

is_tap_changer_Q(*self*)
> Determines whether branch is tap-changing transformer that regulates reactive power flow.
>
> > **Returns flag** : {`True`, `False`}

is_tap_changer_v(*self*)
> Determines whether branch is tap-changing transformer that regulates bus voltage magnitude.
>
> > **Returns flag** : {`True`, `False`}

num_periods
> Number of time periods (int).

obj_type
> Object type (int).

outage
> Flag that indicates whehter branch is on outage.

phase
> Transformer phase shift (radians) (float or array).

phase_max
> Transformer phase shift upper limit (radians) (float).

phase_min
> Transformer phase shift lower limit (radians) (float).

ratingA
> Branch thermal rating A (p.u. system base power) (float).

ratingB
> Branch thermal rating B (p.u. system base power) (float).

ratingC
> Branch thermal rating C (p.u. system base power) (float).

ratio
> Transformer tap ratio (float or array).

ratio_max
  Transformer tap ratio upper limit (float).

ratio_min
  Transformer tap ratio lower limit (float).

reg_bus
  Bus whose voltage is regulated by this tap-changing transformer.

sens_P_l_bound
  Objective function sensitivity with respect to active power flow lower bound (float or array).

sens_P_u_bound
  Objective function sensitivity with respect to active power flow upper bound (float or array).

6.5 Generator

6.5.1 Generator Properties

pfnet.**GEN_PROP_ANY**
  Any generator.

pfnet.**GEN_PROP_SLACK**
  Slack generator.

pfnet.**GEN_PROP_REG**
  Generator that regulates a bus voltage magnitude.

pfnet.**GEN_PROP_NOT_REG**
  Generator that does not regulate a bus voltage magnitude.

pfnet.**GEN_PROP_NOT_SLACK**
  Generator that is not a slack generator.

pfnet.**GEN_PROP_NOT_OUT**
  Generator that is not on outage.

pfnet.**GEN_PROP_P_ADJUST**
  Generator that can adjust its active power, e.g., $P_{\min} < P_{\max}$.

6.5.2 Generator Variables

pfnet.**GEN_VAR_P**
  Generator active power.

pfnet.**GEN_VAR_Q**
  Generator reactive power.

6.5.3 Generator Class

class pfnet.**Generator**(*num_periods=1*, *alloc=True*)
  Generator class.

  Parameters alloc : {True, False}

  num_periods : int

P

Generator active power (p.u. system base MVA) (float or array).

P_cost

Active power generation cost ($/hr) (float or array).

P_max

Generator active power upper limit (p.u. system base MVA) (float).

P_min

Generator active power lower limit (p.u. system base MVA) (float).

P_prev

Generator active power during the previous time period (p.u. system base MVA) (float or array).

Q

Generator reactive power (p.u. system base MVA) (float or array).

Q_max

Generator reactive power upper limit (p.u. system base MVA) (float).

Q_min

Generator reactive power lower limit (p.u. system base MVA) (float).

bus

[Bus](#) to which generator is connected.

cost_coeff_Q0

Coefficient for genertion cost function (constant term, units of $/hr).

cost_coeff_Q1

Coefficient for genertion cost function (linear term, units of $/(hr p.u.)).

cost_coeff_Q2

Coefficient for genertion cost function (quadratic term, units of $/(hr p.u.^2)).

dP_max

Generator active power ramping limit (p.u. system base MVA) (float).

has_flags(*self*, *fmask*, *vmask*)

Determines whether the generator has the flags associated with certain quantities set.

> **Parameters fmask** : int (*Flag Types*)
>
> **vmask** : int (*Generator Variables*)
>
> **Returns flag** : {True, False}

index

Generator index (int).

index_P

Index of generator active power variable (int or array).

index_Q

Index of generator reactive power variable (int or array).

is_P_adjustable(*self*)

Determines whether generator has adjustable active power.

> **Returns flag** : {True, False}

is_equal(*self*, *other*)

Determines whether generator is equal to given generator.

> **Parameters other** : `Generator`

is_on_outage (*self*)
> Determines whether generator in on outage.

> > **Returns flag** : {`True`, `False`}

is_regulator (*self*)
> Determines whether generator provides voltage regulation.

> > **Returns flag** : {`True`, `False`}

is_slack (*self*)
> Determines whether generator is slack.

> > **Returns flag** : {`True`, `False`}

num_periods
> Number of time periods (int).

obj_type
> Object type (int).

outage
> Flag that indicates whehter generator is on outage.

reg_bus
> `Bus` whose voltage is regulated by this generator.

sens_P_l_bound
> Objective function sensitivity with respect to active power lower bound (float or array).

sens_P_u_bound
> Objective function sensitivity with respect to active power upper bound (float or array).

set_P (*self*, *P*, *t=0*)
> ” Sets active power.

> > **Parameters P** : float

> > > **t = int** :

set_Q (*self*, *Q*, *t=0*)
> ” Sets reactive power.

> > **Parameters Q** : float

> > > **t = int** :

6.6 Shunt

6.6.1 Shunt Properties

pfnet.**SHUNT_PROP_ANY**
> Any shunt.

pfnet.**SHUNT_PROP_SWITCHED_V**
> Switched shunt devices that regulates a bus voltage magnitude.

6.6.2 Shunt Variables

pfnet.**SHUNT_VAR_SUSC**
    Switched shunt susceptance.

pfnet.**SHUNT_VAR_SUSC_DEV**
    Switched shunt susceptance deviations from current point.

6.6.3 Shunt Class

class pfnet.**Shunt**(*num_periods=1*, *alloc=True*)
    Shunt class.

> **Parameters alloc** : {`True`, `False`}
>
>> **num_periods** : int

b
    Shunt susceptance (p.u.) (float or array).

b_max
    Shunt susceptance upper limit (p.u.) (float).

b_min
    Shunt susceptance lower limit (p.u.) (float).

bus
    [Bus](#) to which the shunt devices is connected.

g
    Shunt conductance (p.u.) (float).

has_flags(*self*, *fmask*, *vmask*)
    Determines whether the shunt devices has flags associated with certain quantities set.

> **Parameters fmask** : int (*Flag Types*)
>
>> **vmask** : int (*Bus Variables*)
>
> **Returns flag** : {`True`, `False`}

index
    Shunt index (int).

index_b
    Index of shunt susceptance variable (int or array).

index_y
    Index of shunt susceptance positive deviation variable (int or array).

index_z
    Index of shunt susceptance negative deviation variable (int or array).

is_fixed(*self*)
    Determines whether the shunt device is fixed (as opposed to switched).

> **Returns flag** : {`True`, `False`}

is_switched_v(*self*)
    Determines whether the shunt is switchable and regulates bus voltage magnitude.

> **Returns flag** : {`True`, `False`}

> **num_periods**
>> Number of time periods (int).

> **obj_type**
>> Object type (int).

> **reg_bus**
>> Bus whose voltage magnitude is regulated by this shunt device.

6.7 Load

6.7.1 Load Properties

pfnet.**LOAD_PROP_ANY**
> Any load.

pfnet.**LOAD_PROP_P_ADJUST**
> Load that can adjust its active power, e.g., $P_{\min} < P_{\max}$.

6.7.2 Load Variables

pfnet.**LOAD_VAR_P**
> Load active power.

6.7.3 Load Class

class pfnet.**Load**(*num_periods=1*, *alloc=True*)
> Load class.

>> **Parameters alloc** : {True, False}

>>> **num_periods** : int

> **P**
>> Load active power (p.u. system base MVA) (float or array).

> **P_max**
>> Load active power upper limit (p.u. system base MVA) (float).

> **P_min**
>> Load active power lower limit (p.u. system base MVA) (float).

> **P_util**
>> Active power load utility ($/hr) (float or array).

> **Q**
>> Load reactive power (p.u. system base MVA) (float or array).

> **bus**
>> Bus to which load is connected.

> **has_flags**(*self*, *fmask*, *vmask*)
>> Determines whether the load has the flags associated with certain quantities set.

>>> **Parameters fmask** : int (*Flag Types*)

>>>> **vmask** : int (*Load Variables*)

> **Returns flag** : {`True`, `False`}

index
> Load index (int).

index_P
> Index of load active power variable (int or array).

is_P_adjustable(*self*)
> Determines whether the load has adjustable active power.
>
> > **Returns flag** : {`True`, `False`}

num_periods
> Number of time periods (int).

obj_type
> Object type (int).

sens_P_l_bound
> Objective function sensitivity with respect to active power lower bound (float or array).

sens_P_u_bound
> Objective function sensitivity with respect to active power upper bound (float or array).

set_P(*self*, *P*, *t=0*)
> " Sets active power.
>
> > **Parameters P** : float
> >
> > > **t = int** :

set_Q(*self*, *Q*, *t=0*)
> " Sets reactive power.
>
> > **Parameters Q** : float
> >
> > > **t = int** :

util_coeff_Q0
> Coefficient for consumption utility function (constant term, units of $/hr).

util_coeff_Q1
> Coefficient for consumption utility function (linear term, units of $/(hr p.u.)).

util_coeff_Q2
> Coefficient for consumption utility function (quadratic term, units of $/(hr p.u.^2)).

6.8 Variable Generator

6.8.1 Variable Generator Properties

pfnet.**VARGEN_PROP_ANY**
> Any variable generator.

6.8.2 Variable Generator Variables

pfnet.**VARGEN_VAR_P**
> Variable generator active power.

pfnet.**VARGEN_VAR_Q**
>   Variable generator reactive power.

6.8.3 Variable Generator Class

class pfnet.**VarGenerator**(*num_periods=1*, *alloc=True*)
>   Variable generator class.

>>   **Parameters alloc** : {True, False}

>>>   **num_periods** : int

>   **P**
>>   Variable generator active power (p.u. system base MVA) (float or array).

>   **P_max**
>>   Variable generator active power upper limit (p.u. system base MVA) (float).

>   **P_min**
>>   Variable generator active power lower limit (p.u. system base MVA) (float).

>   **P_std**
>>   Variable generator active power standard deviation (p.u. system base MVA) (float or array).

>   **Q**
>>   Variable generator reactive power (p.u. system base MVA) (float or array).

>   **Q_max**
>>   Variable generator maximum reactive power (p.u. system base MVA) (float).

>   **Q_min**
>>   Variable generator minimum reactive power (p.u. system base MVA) (float).

>   **bus**
>>   Bus to which variable generator is connected.

>   **has_flags**(*self*, *fmask*, *vmask*)
>>   Determines whether the variable generator has the flags associated with certain quantities set.

>>>   **Parameters fmask** : int (*Flag Types*)

>>>>   **vmask** : int (*Variable Generator Variables*)

>>>   **Returns flag** : {True, False}

>   **index**
>>   Variable generator index (int).

>   **index_P**
>>   Index of variable generator active power variable (int or array).

>   **index_Q**
>>   Index of variable generator reactive power variable (int or array).

>   **name**
>>   Variable generator name (string).

>   **num_periods**
>>   Number of time periods (int).

>   **obj_type**
>>   Object type (int).

set_P (*self*, *P*, *t=0*)
    ” Sets active power.

> **Parameters P** : float
>
> > **t = int** :

set_P_std (*self*, *P*, *t=0*)
    ” Sets active power standard deviation.

> **Parameters P** : float
>
> > **t = int** :

set_Q (*self*, *Q*, *t=0*)
    ” Sets reactive power.

> **Parameters Q** : float
>
> > **t = int** :

6.9 Battery

6.9.1 Battery Properties

pfnet.**BAT_PROP_ANY**
    Any battery.

6.9.2 Battery Variables

pfnet.**BAT_VAR_P**
    Battery charging/discharging power.

pfnet.**BAT_VAR_E**
    Battery energy level.

6.9.3 Battery Class

class pfnet.**Battery** (*num_periods=1*, *alloc=True*)
    Battery class.

> **Parameters alloc** : {True, False}
>
> > **num_periods** : int

E
    Battery energy level at the beginning of a period (p.u. system base MVA times time unit) (float or array).

E_final
    Battery energy level at the end of the last period (p.u. system base MVA times time unit) (float).

E_init
    Initial battery energy level (p.u. system base MVA times time unit) (float).

E_max
    Battery energy level upper limit (p.u. system base MVA times time unit) (float).

P
> Battery charging power (p.u. system base MVA) (float or array).

P_max
> Battery charging power upper limit (p.u. system base MVA) (float).

P_min
> Battery charging power lower limit (p.u. system base MVA) (float).

bus
> [Bus](#) to which battery is connected.

eta_c
> Battery charging efficiency (unitless) (float).

eta_d
> Battery discharging efficiency (unitless) (float).

has_flags (*self*, *fmask*, *vmask*)
> Determines whether the battery has the flags associated with certain quantities set.

> > **Parameters fmask** : int (*Flag Types*)

> > > **vmask** : int (*Battery Variables*)

> > **Returns flag** : {True, False}

index
> Battery index (int).

index_E
> Index of battery energy level variable (int or array).

index_Pc
> Index of battery charging power variable (int or array).

index_Pd
> Index of battery discharging power variable (int or array).

num_periods
> Number of time periods (int).

obj_type
> Object type (int).

set_E (*self*, *E*, *t=0*)
> Sets battery energy level.

> > **Parameters E** : float

> > > **t** : int

set_P (*self*, *P*, *t=0*)
> Sets battery charging power.

> > **Parameters P** : float

> > > **t** : int

6.10 Network

6.10.1 Component Types

pfnet.**OBJ_ALL**
> All components.

pfnet.**OBJ_BUS**
> Bus.

pfnet.**OBJ_GEN**
> Generator.

pfnet.**OBJ_BRANCH**
> Branch.

pfnet.**OBJ_SHUNT**
> Shunt device.

pfnet.**OBJ_LOAD**
> Load.

pfnet.**OBJ_VARGEN**
> Variable generator (solar, wind, etc).

pfnet.**OBJ_BAT**
> Battery.

pfnet.**OBJ_UNKNOWN**
> Unknown network component.

6.10.2 Flag Types

pfnet.**FLAG_VARS**
> For selecting quantities to be variables.

pfnet.**FLAG_FIXED**
> For selecting variables to be fixed.

pfnet.**FLAG_BOUNDED**
> For selecting variables to be bounded.

pfnet.**FLAG_SPARSE**
> For selecting control adjustments to be sparse.

6.10.3 Flag Short-Cuts

pfnet.**ALL_VARS**
> For selecting all quantifies of a network component to be variables.

6.10.4 Variable Value Codes

pfnet.**CURRENT**
> Current variable value.

pfnet.**UPPER_LIMIT**
> Upper limit of variable.

pfnet.**LOWER_LIMIT**
> Lower limit of variable.

6.10.5 Network Class

class pfnet.**Network**(*num_periods=1*, *alloc=True*)
> Network class.

>> **Parameters alloc** : {True, False}

>>> **num_periods** : int

add_vargens(*self*, *buses*, *penetration*, *uncertainty*, *corr_radius*, *corr_value*)
> Adds variable generators to the network.

>> **Parameters buses** : list of Buses

>>> **penetration** : float

>>>> percentage

>>> **uncertainty** : float

>>>> percentage

>>> **corr_radius** : int

>>>> number of branches

>>> **corr_value** : float

>>>> correlation coefficient

adjust_generators(*self*)
> Adjusts powers of slack and regulator generators connected to or regulating the same bus to correct generator participations without modifying the total power injected.

base_power
> System base power (MVA) (float).

batteries
> List of network batteries (list).

branches
> List of network branches (list).

bus_P_mis
> Largest bus active power mismatch in the network (MW) (float or array).

bus_Q_mis
> Largest bus reactive power mismatch in the network (MVAr) (float or array).

bus_v_max
> Maximum bus voltage magnitude (p.u.) (float or array).

bus_v_min
> Minimum bus voltage magnitude (p.u.) (float or array).

bus_v_vio
> Maximum bus voltage magnitude limit violation (p.u.) (float or array).

buses
 List of network `buses` (list).

clear_error (*self*)
 Clear error flag and message string.

clear_flags (*self*)
 Clears all the flags of all the network components.

clear_properties (*self*)
 Clears all the network properties.

clear_sensitivities (*self*)
 Clears all sensitivity information.

create_sorted_bus_list (*self*, *sort_by*, *t=0*)
 Creates list of buses sorted in descending order according to a specific quantity.

> **Parameters sort_by** : int (*Bus Sensitivities*, *Bus Power Mismatches*).
>
> > **t** : int
>
> **Returns buses** : list of `Buses`

create_vargen_P_sigma (*self*, *spread*, *corr*)
 Creates covariance matrix (lower triangular part) for variable vargen active powers.

> **Parameters spead** : int
>
> > Determines correlation neighborhood in terms of number of edges.
>
> > **corr** : float
>
> > Desired correlation coefficient for neighboring vargens.
>
> **Returns sigma** : `coo_matrix`

gen_P_cost
 Total active power generation cost ($/hr) (float or array).

gen_P_vio
 Largest generator active power limit violation (MW) (float or array).

gen_Q_vio
 Largest generator reactive power limit violation (MVAr) (float or array).

gen_v_dev
 Largest voltage magnitude deviation from set point of bus regulated by generator (p.u.) (float or array).

generators
 List of network `generators` (list).

get_bat (*self*, *index*)
 Gets battery with the given index.

> **Parameters index** : int
>
> **Returns bat** : `Battery`

get_branch (*self*, *index*)
 Gets branch with the given index.

> **Parameters index** : int
>
> **Returns branch** : `Branch`

get_bus (*self*, *index*)
> Gets bus with the given index.

>> **Parameters index** : int

>> **Returns bus** : `Bus`

get_bus_by_name (*self*, *name*)
> Gets bus with the given name.

>> **Parameters name** : string

>> **Returns bus** : `Bus`

get_bus_by_number (*self*, *number*)
> Gets bus with the given number.

>> **Parameters number** : int

>> **Returns bus** : `Bus`

get_gen (*self*, *index*)
> Gets generator with the given index.

>> **Parameters index** : int

>> **Returns gen** : `Generator`

get_gen_buses (*self*)
> Gets list of buses where generators are connected.

>> **Returns buses** : list

get_load (*self*, *index*)
> Gets load with the given index.

>> **Parameters index** : int

>> **Returns gen** : `Load`

get_load_buses (*self*)
> Gets list of buses where loads are connected.

>> **Returns buses** : list

get_num_P_adjust_gens (*self*)
> Gets number of generators in the network that have adjustable active powers.

>> **Returns num** : int

get_num_P_adjust_loads (*self*)
> Gets number of loads in the network that have adjustable active powers.

>> **Returns num** : int

get_num_batteries (*self*)
> Gets number of batteries in the network.

>> **Returns num** : int

get_num_branches (*self*)
> Gets number of branches in the network.

>> **Returns num** : int

get_num_branches_not_on_outage (*self*)
> Gets number of branches in the network that are not on outage.

Returns num : int

get_num_buses(*self*)
Gets number of buses in the network.

Returns num : int

get_num_buses_reg_by_gen(*self*)
Gets number of buses whose voltage magnitudes are regulated by generators.

Returns num : int

get_num_buses_reg_by_shunt(*self*, *only=False*)
Gets number of buses whose voltage magnitudes are regulated by switched shunt devices.

Returns num : int

get_num_buses_reg_by_tran(*self*, *only=False*)
Gets number of buses whose voltage magnitudes are regulated by tap-changing transformers.

Returns num : int

get_num_fixed_shunts(*self*)
Gets number of fixed shunts in the network.

Returns num : int

get_num_fixed_trans(*self*)
Gets number of fixed transformers in the network.

Returns num : int

get_num_generators(*self*)
Gets number of generators in the network.

Returns num : int

get_num_gens_not_on_outage(*self*)
Gets number of generators in the network that are not on outage.

Returns num : int

get_num_lines(*self*)
Gets number of transmission lines in the network.

Returns num : int

get_num_loads(*self*)
Gets number of loads in the network.

Returns num : int

get_num_phase_shifters(*self*)
Gets number of phase-shifting transformers in the network.

Returns num : int

get_num_reg_gens(*self*)
Gets number generators in the network that provide voltage regulation.

Returns num : int

get_num_shunts(*self*)
Gets number of shunts in the network.

Returns num : int

get_num_slack_buses(*self*)
 Gets number of slack buses in the network.

   Returns num : int

get_num_slack_gens(*self*)
 Gets number of slack generators in the network.

   Returns num : int

get_num_switched_shunts(*self*)
 Gets number of switched shunts in the network.

   Returns num : int

get_num_tap_changers(*self*)
 Gets number of tap-changing transformers in the network.

   Returns num : int

get_num_tap_changers_Q(*self*)
 Gets number of tap-changing transformers in the network that regulate reactive flows.

   Returns num : int

get_num_tap_changers_v(*self*)
 Gets number of tap-changing transformers in the network that regulate voltage magnitudes.

   Returns num : int

get_num_var_generators(*self*)
 Gets number of variable generators in the network.

   Returns num : int

get_properties(*self*)
 Gets network properties.

   Returns properties : dict

get_shunt(*self*, *index*)
 Gets shunt with the given index.

   Parameters index : int

   Returns gen : Shunt

get_var_projection(*self*, *obj_type*, *var*, *t_start=0*, *t_end=None*)
 Gets projection matrix for specific object variables.

   Parameters obj_type : int (*Component Types*)

      var : int (*Bus Variables*, *Branch Variables*, *Generator Variables*, *Shunt Variables*, *Load Variables*, *Variable Generator Variables*, *Battery Variables*)

      t_start : int

      t_end : int (inclusive)

get_var_values(*self*, *code=CURRENT*)
 Gets network variable values.

   Parameters code : int (See var values)

   Returns values : ndarray

get_vargen(*self*, *index*)
    Gets variable generator with the given index.

    Parameters index : int

    Returns vargen : `VarGenerator`

get_vargen_by_name(*self*, *name*)
    Gets vargen with the given name.

    Parameters name : string

    Returns vargen : `VarGenerator`

has_error(*self*)
    Indicates whether the network has the error flag set due to an invalid operation.

load(*self*, *filename*, *output_level=0*)
    Loads a network data contained in a specific file.

    Parameters filename : string

        output_level : int

load_P_util
    Total active power consumption utility ($/hr) (float or array).

load_P_vio
    Largest load active power limit violation (MW) (float or array).

loads
    List of network `loads` (list).

num_actions
    Number of control adjustments (int or array).

num_batteries
    Number of batteries in the network (int).

num_bounded
    Number of network quantities that have been set to bounded (int).

num_branches
    Number of branches in the network (int).

num_buses
    Number of buses in the network (int).

num_fixed
    Number of network quantities that have been set to fixed (int).

num_generators
    Number of generators in the network (int).

num_loads
    Number of loads in the network (int).

num_periods
    Number of time periods (int).

num_shunts
    Number of shunt devices in the network (int).

num_sparse
    Number of network control quantities that have been set to sparse (int).

num_var_generators
> Number of variable generators in the network (int).

num_vars
> Number of network quantities that have been set to variable (int).

set_flags (*self, obj_type, flags, props, vals*)
> Sets flags of network components with specific properties.
>
>> **Parameters obj_type** : int (*Component Types*)
>>
>> **flags** : int or list (*Flag Types*)
>>
>> **props** : int or list (*Bus Properties*, *Branch Properties*, *Generator Properties*, *Shunt Properties*, *Load Properties*, *Variable Generator Properties*, *Battery Properties*)
>>
>> **vals** : int or list (*Bus Variables*, *Branch Variables*, *Generator Variables*, *Shunt Variables*, *Load Variables*, *Variable Generator Variables*, *Battery Variables*)

set_flags_of_component (*self, obj, flags, vals*)
> Sets flags of network components with specific properties.
>
>> **Parameters obj** : `Bus`, `Branch`, `Generator`, `Load`, `Shunt`, `VarGenerator`, `Battery`
>>
>> **flags** : int or list (*Flag Types*)
>>
>> **vals** : int or list (*Bus Variables*, *Branch Variables*, *Generator Variables*, *Shunt Variables*, *Load Variables*, *Variable Generator Variables*, *Battery Variables*)

set_var_values (*self, values*)
> Sets network variable values.
>
>> **Parameters values** : `ndarray`

show_buses (*self, number, sort_by, t=0*)
> Shows information about the most relevant network buses sorted by a specific quantity.
>
>> **Parameters number** : int
>>
>> **sort_by** : int (*Bus Sensitivities*, *Bus Power Mismatches*)
>>
>> **t** : int (time period)

show_components (*self*)
> Shows information about the number of network components of each type.

show_properties (*self, t=0*)
> Shows information about the state of the network component quantities.
>
>> **Parameters t** : int (time period)

shunt_b_vio
> Largest switched shunt susceptance limit violation (p.u.) (float or array).

shunt_v_vio
> Largest voltage magnitude band violation of voltage regulated by switched shunt device (p.u.) (float or array).

shunts
> List of network `shunts` (list).

tran_p_vio
> Largest transformer phase shift limit violation (float or array).

tran_r_vio
> Largest transformer tap ratio limit violation (float or array).

tran_v_vio
>   Largest voltage magnitude band violation of voltage regulated by transformer (p.u.) (float or array).

update_properties(*self*, *values=None*)
>   Re-computes the network properties using the given values of the network variables. If no values are given, then the current values of the network variables are used.

>>   **Parameters values** : `ndarray`

update_set_points(*self*)
>   Updates voltage magnitude set points of gen-regulated buses to be equal to the bus voltage magnitudes.

var_generators
>   List of network `variable generators` (list).

vargen_corr_radius
>   Correlation radius of variable generators (number of edges).

vargen_corr_value
>   Correlation value (coefficient) of variable generators.

6.11 Contingency

class pfnet.**Contingency**(*gens=None*, *branches=None*, *alloc=True*)
>   Contingency class.

>>   **Parameters gens** : list or `Generators`

>>   **branches** : list `Branchs`

>>   **alloc** : {`True`, `False`}

add_branch_outage(*self*, *br*)
>   Adds branch outage to contingency.

>>   **Parameters br** : `Branch`

add_gen_outage(*self*, *gen*)
>   Adds generator outage to contingency.

>>   **Parameters gen** : `Generator`

apply(*self*)
>   Applies outages that characterize contingency.

clear(*self*)
>   Clears outages that characterize contingency.

has_branch_outage(*self*, *br*)
>   Determines whether contingency specifies the given branch as being on outage.

>>   **Parameters branch** : `Branch`

>>   **Returns result** : {`True`, `False`}

has_gen_outage(*self*, *gen*)
>   Determines whether contingency specifies the given generator as being on outage.

>>   **Parameters gen** : `Generator`

>>   **Returns result** : {`True`, `False`}

num_branch_outages
    Number of branch outages.

num_gen_outages
    Number of generator outages.

show (*self*)
    Shows contingency information.

6.12 Graph

class `pfnet.Graph` (*net*, *alloc=True*)
    Graph class.

        Parameters net : `Network`

            alloc : {`True`, `False`}

clear_error (*self*)
    Clear error flag and message string.

color_nodes_by_mismatch (*self*, *mis_type*, *t=0*)
    Colors the graphs nodes according to their power mismatch.

        Parameters mis_type : int (*Bus Power Mismatches*)

            t : int

color_nodes_by_sensitivity (*self*, *sens_type*, *t=0*)
    Colors the graphs nodes according to their sensitivity.

        Parameters sens_type : int (*Bus Sensitivities*)

            t : int

has_error (*self*)
    Indicates whether the graph has the error flag set due to an invalid operation.

has_viz (*self*)
    Determines whether graph has visualization capabilities.

        Returns flag : {`True`, `False`}

set_edges_property (*self*, *prop*, *value*)
    Sets property of edges. See Graphviz documentation.

        Parameters prop : string

            value : string

set_layout (*self*)
    Determines and saves a layout for the graph nodes.

set_node_property (*self*, *bus*, *prop*, *value*)
    Sets property of node. See Graphviz documentation.

        Parameters bus : `Bus`

            prop : string

            value : string

set_nodes_property (*self*, *prop*, *value*)
    Sets property of nodes. See Graphviz documentation.

> > > **Parameters** **prop** : string
> > >
> > > **value** : string

> **view**(*self*, *inline=False*)
>
> > Displays the graph.

> **write**(*self*, *format*, *filename*)
>
> > Writes the graph to a file.

> > > **Parameters** **format** : string (Graphviz output formats)
> > >
> > > **filename** : string

6.13 Function

6.13.1 Function Types

pfnet.**FUNC_TYPE_UNKNOWN**
> Unknown function.

pfnet.**FUNC_TYPE_REG_VMAG**
> Bus voltage magnitude regularization.

pfnet.**FUNC_TYPE_SLIM_VMAG**
> Bus voltage magnitude soft limits penalty.

pfnet.**FUNC_TYPE_REG_VANG**
> Bus voltage angle regularization.

pfnet.**FUNC_TYPE_REG_PQ**
> Generator active and reactive power regularization.

pfnet.**FUNC_TYPE_GEN_COST**
> Active power generation cost.

pfnet.**FUNC_TYPE_NETCON_COST**
> Net power consumption cost.

pfnet.**FUNC_TYPE_LOAD_UTIL**
> Active power consumption utility.

pfnet.**FUNC_TYPE_REG_RATIO**
> Transformer tap ratio regularization.

pfnet.**FUNC_TYPE_REG_PHASE**
> Transformer phase shift regularization.

pfnet.**FUNC_TYPE_REG_SUSC**
> Switched shunt susceptance regularization.

pfnet.**FUNC_TYPE_SP_CONTROLS**
> Sparsity-inducing penalty for control adjustments.

6.13.2 Function Class

class pfnet.**Function**(*int type*, *float weight*, *Network net*, *alloc=True*)
> Function class.

> > **Parameters type** : int (*Function Types*)
>
> > **weight** : float
>
> > **net** : `Network`
>
> > **alloc** : {`True`, `False`}

Hcounter
> Number of nonzero entries in Hessian matrix (int).

Hphi
> Function Hessian matrix (only the lower triangular part) (`coo_matrix`).

analyze(*self*)
> Analyzes function and allocates required vectors and matrices.

clear_error(*self*)
> Clears internal error flag.

del_matvec(*self*)
> Deletes matrices and vectors associated with this function.

eval(*self*, *var_values*)
> Evaluates function value, gradient, and Hessian using the given variable values.

> > **Parameters var_values** : `ndarray`

gphi
> Function gradient vector (`ndarray`).

phi
> Function value (float).

type
> Function type (int).

update_network(*self*)
> Updates internal arrays to be compatible with any network changes.

weight
> Function weight (float).

6.14 Constraint

6.14.1 Constraint Types

pfnet.**CONSTR_TYPE_PF**
> Constraint for enforcing AC power balance at every bus of the network.

pfnet.**CONSTR_TYPE_DCPF**
> Constraint for enforcing DC power balance at every bus of the network.

pfnet.**CONSTR_TYPE_LINPF**
> Constraint for enforcing linearized power balance at every bus of the network.

pfnet.**CONSTR_TYPE_FIX**
> Constraint for fixing a subset of variables to their current value.

pfnet.**CONSTR_TYPE_BOUND**
> Constraint for forcing a subset of variables to be within their bounds (nonlinear).

pfnet.**CONSTR_TYPE_LBOUND**
    Constraint for forcing a subset of variables to be within their bounds (linear).

pfnet.**CONSTR_TYPE_PAR_GEN_P**
    Constraint for enforcing generator active power participations.

pfnet.**CONSTR_TYPE_PAR_GEN_Q**
    Constraint for enforcing generator reactive power participations.

pfnet.**CONSTR_TYPE_REG_GEN**
    Constraint for enforcing voltage set point regulation by generators.

pfnet.**CONSTR_TYPE_REG_TRAN**
    Constraint for enforcing voltage band regulation by tap-changing transformers.

pfnet.**CONSTR_TYPE_REG_SHUNT**
    Constraint for enforcing voltage band regulation by switched shunt devices.

pfnet.**CONSTR_TYPE_DC_FLOW_LIM**
    Constraint for enforcing DC power flow limits on every branch.

pfnet.**CONSTR_TYPE_GEN_RAMP**
    Constraint for enforcing generator active power ramping limits.

6.14.2 Constraint Class

class pfnet.**Constraint**(*int type*, *Network net*, *alloc=True*)
    Contraint class.

> **Parameters type** : int (*Constraint Types*)
>
> **net** : `Network`
>
> **alloc** : {True, False}

A
    Matrix for linear equality constraints (`coo_matrix`).

Aconstr_index
    Index of linear equality constraint (int).

Acounter
    Number of nonzero entries in the matrix of linear equality constraints (int).

G
    Matrix for linear inequality constraints (`coo_matrix`).

Gconstr_index
    Index of linear ineqquality constraint (int).

Gcounter
    Number of nonzero entries in the matrix of linear inequality constraints (int).

H_combined
    Linear combination of Hessian matrices of individual nonlinear equality constraints (only the lower triangular part) (`coo_matrix`).

J
    Jacobian matrix of nonlinear equality constraints (`coo_matrix`).

Jconstr_index
    Index of nonlinear equality constraint (int).

Jcounter
>   Number of nonzero entries in the Jacobian matrix of the nonlinear equality constraints (int).

analyze(*self*)
>   Analyzes constraint and allocates required vectors and matrices.

b
>   Right-hand side vector of linear equality constraints (`ndarray`).

clear_error(*self*)
>   Clears internal error flag.

combine_H(*self*, *coeff*, *ensure_psd=False*)
>   Forms and saves a linear combination of the individual constraint Hessians.
>
>   > **Parameters coeff** : `ndarray`
>   >
>   > **ensure_psd** : {`True`, `False`}

del_matvec(*self*)
>   Deletes matrices and vectors associated with this constraint.

eval(*self*, *var_values*)
>   Evaluates constraint violations, Jacobian, and individual Hessian matrices.
>
>   > **Parameters var_values** : `ndarray`

f
>   Vector of nonlinear equality constraint violations (`ndarray`).

get_H_single(*self*, *i*)
>   Gets the Hessian matrix (only lower triangular part) of an individual constraint.
>
>   > **Parameters i** : int
>   >
>   > **Returns H** : `coo_matrix`

l
>   Lower bound vector of linear inequality constraints (`ndarray`).

store_sensitivities(*self*, *sA*, *sf*, *sGu*, *sGl*)
>   Stores Lagrange multiplier estimates of the constraints in the power network components.
>
>   > **Parameters sA** : `ndarray`
>   >
>   > > sensitivities for linear equality constraints ($Ax = b$)
>   >
>   > **sf** : `ndarray`
>   >
>   > > sensitivities for nonlinear equality constraints ($f(x) = 0$)
>   >
>   > **sGu** : `ndarray`
>   >
>   > > sensitivities for linear inequality constraints ($Gx \leq u$)
>   >
>   > **sGl** : `ndarray`
>   >
>   > > sensitivities for linear inequality constraints ($l \leq Gx$)

type
>   Constraint type (*Constraint Types*) (int).

u
>   Upper bound vector of linear inequality constraints (`ndarray`).

update_network(*self*)
>   Updates internal arrays to be compatible with any network changes.

6.15 Optimization Problem

6.15.1 Problem Class

class pfnet.**Problem**

Optimization problem class.

A

Constraint matrix of linear equality constraints (`coo_matrix`).

G

Constraint matrix of linear inequality constraints (`coo_matrix`).

H_combined

Linear combination of Hessian matrices of individual nonlinear equality constraints (only the lower triangular part) (`coo_matrix`).

Hphi

Objective function Hessian matrix (only the lower triangular part) (`coo_matrix`).

J

Jacobian matrix of the nonlinear equality constraints (`coo_matrix`).

add_constraint (*self*, *ctype*)

Adds constraint to optimization problem.

> **Parameters ctype** : int (*Constraint Types*)

add_function (*self*, *ftype*, *weight*)

Adds function to optimization problem objective.

> **Parameters ftype** : int (*Function Types*)
>
> > **weight** : float

analyze (*self*)

Analyzes function and constraint structures and allocates required vectors and matrices.

b

Right hand side vectors of the linear equality constraints (`ndarray`).

clear (*self*)

Resets optimization problem data.

combine_H (*self*, *coeff*, *ensure_psd*)

Forms and saves a linear combination of the individual constraint Hessians.

> **Parameters coeff** : `ndarray`
>
> > **ensure_psd** : {True, False}

constraints

List of `constraints` of this optimization problem (list).

eval (*self*, *var_values*)

Evaluates objective function and constraints as well as their first and second derivatives using the given variable values.

> **Parameters var_values** : `ndarray`

f

Vector of nonlinear equality constraints violations (`ndarray`).

find_constraint (*self*, *type*)
    Finds constraint of give type among the constraints of this optimization problem.

>   **Parameters type** : int (*Constraint Types*)

functions
    List of `functions` that form the objective function of this optimization problem (list).

get_init_point (*self*)
    Gets initial solution estimate from the current value of the network variables.

>   **Returns point** : `ndarray`

get_lower_limits (*self*)
    Gets vector of lower limits for the network variables.

>   **Returns limits** : `ndarray`

get_network (*self*)
    Gets the power network associated with this optimization problem.

get_upper_limits (*self*)
    Gets vector of upper limits for the network variables.

>   **Returns limits** : `ndarray`

gphi
    Objective function gradient vector (`ndarray`).

l
    Lower bound for linear inequality constraints (`ndarray`).

lam
    Initial dual point (`ndarray`).

network
    Power network associated with this optimization problem (`Network`).

nu
    Initial dual point (`ndarray`).

phi
    Objective function value (float).

set_network (*self*, *net*)
    Sets the power network associated with this optimization problem.

show (*self*)
    Shows information about this optimization problem.

store_sensitivities (*self*, *sA*, *sf*, *sGu*, *sGl*)
    Stores Lagrange multiplier estimates of the constraints in the power network components.

>   **Parameters sA** : `ndarray`
>
>   sensitivities for linear equality constraints ($Ax = b$)
>
>   **sf** : `ndarray`
>
>   sensitivities for nonlinear equality constraints ($f(x) = 0$)
>
>   **sGu** : `ndarray`
>
>   sensitivities for linear inequality constraints ($Gx \leq u$)
>
>   **sGl** : `ndarray`

sensitivities for linear inequality constraints ($l \leq Gx$)

u
> Upper bound for linear inequality constraints (`ndarray`).

update_lin (*self*)
> Updates linear equality constraints.

x
> Initial primal point (`ndarray`).

6.16 References

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

[TTR2015] T. Tinoco De Rubira, *Numerical Optimization and Modeling Techniques for Power System Operations and Planning*. PhD thesis, Stanford University, March 2015.

p

pfnet, 1

p

O

P