Pylon Documentation

Release 0.2

Richard W. Lincoln

March 30, 2009

CONTENTS

CHAPTER

ONE

INTRODUCTION

Pylon is a software package for simulation and analysis of electric power systems and energy markets. It provides `Bus` and `Branch` Python objects for representing power systems in graph form. `Generator` and `Load` objects may be added to a `Bus` objects and define levels of active supply and passive demand.

Subpackages of `pylon` define further functionality.

`pylon.routine` Routines for solving power flow and optimal power flow problems. The routines are translated from MATPOWER and use the sparse matrix types and optimisation routines from CVXOPT.

`pylon.pyreto` Modules for simulating competitive energy trade using reinforcement learning algorithms and artificial neural networks from PyBrain.

`pylon.readwrite` Parsers for a selection of power system data file formats including MATPOWER, PSS/E, and PSAT. Export of data in MATPOWER, CSV and Excel file formats. Reports in ReStructuredText format.

`pylon.test` A comprehensive suite of unit tests.

`pylon.ui` <pylon.ui>‘ Cross-platform, toolkit independent user interfaces via the TraitsGUI package. Interactive, publication quality data plots using Chaco. Graphviz powered interactive 2D graph visualisation using Godot. Plug-ins for the Envisage application framework.

This manual describes how `Network` models may be constructed and the subpackages used in their simulation and analysis. The routines in Pylon are translated from MATPOWER, the user manual for which will likely provide a more useful reference.

CHAPTER

TWO

INSTALLATION

Pylon is a package of Python modules. It needs to be on the PYTHON_PATH environment variable and for some core dependencies to be met for model and solver functionality. Optionally, easily installed additional libraries enable other features.

2.1 Dependencies

2.5 <= Python < 3.0

Traits 3.0 or later Provides Python object attributes with additional characteristics.

CVXOPT 1.0 or later CVXOPT is a free software package for convex optimization based on the Python programming language.

NumPy 1.2 or later NumPy provides additional support for multi-dimentional arrays and matrices.

2.2 Strongly recommended

Pyparsing Pyparsing is a versatile Python module for recursive descent parsing.

PyBrain PyBrain is a modular Machine Learning Library for Python.

iPython Interactive python interpreter.

wxPython Cross-platform GUI toolkit for the Python programming language.

Godot Godot uses Graphviz Xdot output to provide interactive graph visualisation.

2.3 Windows

The Enthought Python Distribution provides the majority of the dependencies for Pylon and is free for academic use. CVXOPT is not included, but comes as a Windows Installer also.

2.4 Setuptools

With Python and setuptools installed, simply:

```
$ easy_install pylon
```

Virtualenv may be used to build a virtual Python environment:

```
$ virtualenv env
$ ./env/bin/easy_install pylon
```

2.5 Installation from source

Extract the gzipped tar file:

```
$ tar xvf pylon-X.X.tar.gz
```

Run the `setup.py` script:

```
$ cd pylon-X.X
$ python setup.py install
```

or:

```
$ python setup.py develop
```

2.6 Working directory

Change in to the source directory and run IPython:

```
$ cd ~/path/to/pylon-X.X
$ ipython
```

Access the `pylon` application programming inteface.

```
In [1]: from pylon.api import Generator, DCOPFRoutine
```

CHAPTER

THREE

POWER SYSTEM MODEL

This chapter describes the Pylon objects that may be used to model a power system. It explains their main features and how they may be associated with one and other.

3.1 Network

class Network()

A `Network` object is a representation of a power system as a graph. It contains a list of `Bus` objects that define the nodes of the graph and a list of `Branch` objects that define the edges.

For convenience, a `Network` provides certain read-only properties, such as `all_generators`, that are used regularly by other modules.

```
In [1]: from pylon import Network
In [2]: network = Network(name="net1", mva_base=100.0)
```

class Bus()

A `Bus` is a node in the power system graph to which `Generator` and `Load` objects may be added.

```
In [1]: from pylon import Network, Bus
In [2]: network = Network()
In [3]: bus = Bus(name="bus1", v_amplitude_guess=1.1)
In [4]: network.buses.append(bus)
```

The objects connected determine the `mode` of the `Bus`, which may be one of three values:

- PQ
  - Default mode for a plain bus.
  - Set when one or more `Load` is present, but no `Generator`.
  - Set when the connected generation has reached one of it's reactive power limits.
  - Active and reactive power are the known variables.
- PV
  - Set when one or more `Generator` is connected to the `Bus` and the `slack` attribute of the `Bus` is not set.
  - Active power and voltage magnitude are the known system variables.
- Slack

- Sometimes known as the "swing" of reference bus.
- Set when one or more `Generator` is connected to the `Bus` and the `slack` attribute is true.
- No variables to be solved for in the power flow solution.

The shunt conductance and susceptance at the bus are specified by the `g_shunt` and `b_shunt` attributes respectively.

For convenience, `Bus` provides read-only properties that return values of total supply and demand of power at the node.

class Branch ()

Transmission lines and transformers are both defined by the `Branch` class which uses a standard pi-circuit model. The `source_bus` and `target_bus` must be specified when creating a `Branch`.

```
In [1]: network = Network()
In [2]: bus1, bus2 = Bus(), Bus()
In [3]: e = Branch(bus1, bus2, r=0.06, x=0.03)
In [4]: network.branches.append(branch)
```

class Generator ()

A `Generator` specifies the voltage magnitude and the active power injected at a node. If reactive power limits are enforced then a `Generator` may switch to fixing active and reactive power at a node if a limit is violated.

```
In [1]: bus = Bus()
In [2]: g = Generator(p=6.0, v_amplitude=1.1)
In [3]: bus.generators.append(g)
In [4]: bus.mode
Out[1]: 'PV'
```

`Generator` objects define the despatchable units for the optimal power flow problem. The `p_max_bid` and `p_min_bid` attributes define the range in which the generator is willing to operate and this must be within the rated capacity of the unit as defined by `p_max` and `p_min`. The cost of the generator with respect to active power is defined using the `cost_coeffs` attribute. This is a triple of floating point values, restricting the definition of cost curves to quadratic functions.

```
In [1]: g = Generator(p_max=6.0, p_min=1.0, cost_coeffs=(0.0, 6.0, 0.0)
                      p_max_bid=5.0, p_min_bid=1.0)
```

class Load ()

A load fixes active and reactive power demand at a the node.

A `Load` may be configured to follow an output profile. The attribute `p_profile` specifies a list of percentages that define how the profile varies between the limits defined by `p_max` and `p_min`. `p_profiled` is a property that uses a cycle iterator to return the next value in the profile sequence each time it is called.

```
In [1]: l = Load(p_min=1.0, p_max=2.0, p_profile=[100, 50])
In [2]: l.p_profiled
Out[1]: 2.0

In [3]: l.p_profiled
Out[2]: 1.5

In [4]: l.p_profiled
Out[3]: 2.0
```

CHAPTER

FOUR

PARSERS

Pylon uses Pyparsing to parse power system data files. The MATPOWER parser is the most robust as it is used in test cases for the routines. Parsers are also available for:

- PSS/E raw files and
- PSAT .m files.

4.1 MATPOWER

Simply:

```
>>> from pylon.readwrite.api import read_matpower
>>> network = read_matpower('/path/to/casefile.m')
```

The parser class is in the API also:

```
>>> from pylon.readwrite.api import MATPOWERReader
>>> reader = MATPOWERReader('/path/to/casefile.m')
>>> network = reader.network
>>> network2 = reader.parse_file('/path/to/casefile2.m')
```

4.2 PTI PSS/E

The PSS/E parser is tested with RAW files from the UKGDS project:

```
>>> from pylon.readwrite.api import read_psse
>>> network = read_psse('ehv3.raw')
```

4.3 CIM RDF/XML

Pylon includes a parser for RDF/XML files with Common Information Model data. The parser builds a `Model` object with a list of instances of the classes available in the `CIM13` package. `Model` is not yet compatible with any of the routines.

The parser can accept XML, Zip, Gzip and bzip2 files:

```
>>> from pylon.readwrite.cim_reader import CIMReader
>>> reader = CIMReader('/path/to/data.xml')
>>> model = reader.parse_file()
>>> model2 = reader.parse_file('/path/to/data.gz')
```

CHAPTER

FIVE

ROUTINES

Pylon includes a selection of routines for solving power flow and optimal power flow problems. The routines are translated from MATPOWER and use the sparse matrix types and optimisation routines from CVXOPT.

5.1 DC Power Flow

The DC power flow routine is made linear by making the assumption that branch losses are negligible, all bus voltages are 1.0 p.u. and that voltage phase angles are small.

CVXOPT provides interfaces to CHOLMOD and UMFPACK, both of which provide routines for solving sets of sparse linear equations. The `routine` attribute of `DCPFRoutine` specifies which library to use and defaults to 'UMFPACK'.

5.2 Newton Power Flow

`NewtonPFRoutine` is a subclass of `ACPFRoutine` that solves the power flow problem using standard Newton's method with a full Jacobian updated each iteration and sparsity maintained throughout. `ACPFRoutine` is a base class with methods common to all power flow routines using an AC formulation.

5.3 DC Optimal Power Flow

The DC formulation of the optimal power flow routine uses the `qp` solver from CVXOPT for solving quadratic programs. Optionally, the `solver` attribute of the routine may be set to 'mosek' if MOSEK version 5 is available.

5.4 AC Optimal Power Flow

The AC optimal power flow routine uses the `cp` solver from CVXOPT to minimise a non-linear objective function that is subject to non-linear constraints. The solver is written in Python and may be found in the `cvxprog.py` module in the CVXOPT distribution.

INDEX