Pylon Documentation

Release 0.4.2

Richard Lincoln

April 13, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

Pylon is a port of MATPOWER to the Python programming language. MATPOWER is a Matlab package for solving power flow and optimal power flow problems.

pylon Defines the `Case`, `Bus`, `Branch` and `Generator` classes and solvers for power flow and optimal power flow problems.

pylon.readwrite Parsers for power system data files with support for MATPOWER, PSS/E, and PSAT. Also, defines case serializers for MATPOWER, PSS/E, CSV and Excel formats. Case reports are available in ReStructuredText format.

pylon.test A comprehensive suite of unit tests.

This manual explains how to install Pylon and provides a series of tutorials that show how to solve power flow and optimal power problems. Pylon follows the design of MATPOWER closely and the MATPOWER user manual will likely provide a useful reference.

CHAPTER

TWO

INSTALLATION

Pylon is a package of Python modules that need to be placed on the `PYTHON_PATH`.

2.1 Dependencies

Python 2.5 or 2.6

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

SciPy 0.7 or later Packages for mathematics, science, and engineering

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

2.2 Recommended

scikits.umfpack Wrappers of UMFPACK sparse direct solver to SciPy.

2.3 Setuptools

With Python and setuptools installed, simply:

```
$ easy_install pylon
```

Users without root access may use Virtualenv to build a virtual Python environment:

```
$ virtualenv python26
$ ./python26/bin/easy_install pylon
```

To upgrade to a newer version:

```
$ easy_install -U pylon
```

2.4 Installation from source

Run the `setup.py` script:

```
$ python setup.py install
```

or:

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

2.5 Working directory

Change in to the source directory and run IPython:

```
$ cd ~/path/to/pylon-0.4.2
$ ipython
```

Access the `pylon` application programming inteface.

```
In [1]: from pylon import Case, OPF
```

CHAPTER

THREE

TUTORIAL

3.1 Power Flow

Import "sys" so the report can be written to stdout.

```
import sys
```

Import the logging module

```
import logging
```

and set up a basic configuration.

```
logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)
```

The "pylon" package contains classes for defining a power system model and power flow solvers.

```
from pylon import \
    Case, Bus, Branch, Generator, NewtonPF, FastDecoupledPF, REFERENCE
```

Start by building up a one branch case with two generators

```
bus1 = Bus(type=REFERENCE)
g1 = Generator(bus1, p=80.0, q=10.0)
```

and fixed load at the other.

```
bus2 = Bus(p_demand=60.0, q_demand=4.0)
g2 = Generator(bus2, p=20.0, q=0.0)
```

Connect the two buses

```
line = Branch(bus1, bus2, r=0.05, x=0.01)
```

and add it all to a new case.

```
case = Case(buses=[bus1, bus2], branches=[line], generators=[g1, g2])
```

Choose to solve using either Fast Decoupled method

```
solver = FastDecoupledPF(case)
```

or Newton's method

```
solver = NewtonPF(case)
```

and then call the solver.

```
solver.solve()
```

Write the case out to view the results.

```
case.save_rst(sys.stdout)
```

3.2 Optimal Power Flow

This tutorial provides a guide for solving an Optimal Power Flow problem using Pylon.

First import the necessary components from Pylon.

```
from pylon import Case, Bus, Branch, Generator, OPF, REFERENCE
```

Import "sys" for writing to stdout.

```
import sys
```

Import the logging module

```
import logging
```

and set up a basic configuration.

```
logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)
```

Create two generators, specifying their marginal cost.

```
bus1 = Bus(p_demand=100.0, type=REFERENCE)
g1 = Generator(bus1, p_min=0.0, p_max=80.0, p_cost=[(0., 0.), (80., 4800.)])
bus2 = Bus()
g2 = Generator(bus2, p_min=0.0, p_max=60.0, p_cost=[(0., 0.), (60., 4500.)])
```

Connect the two generator buses

```
line = Branch(bus1, bus2, r=0.05, x=0.25, b=0.06)
```

and add it all to a case.

```
case = Case(buses=[bus1, bus2], branches=[line], generators=[g1, g2])
```

Non-linear AC optimal power flow

```
dc = False
```

or linearised DC optimal power flow may be selected.

```
dc = True
```

Pass the case to the OPF routine and solve.

```
OPF(case, dc).solve()
```

View the results as ReStructuredText.

```
case.save_rst(sys.stdout)
```

CHAPTER

FOUR

API

4.1 `pylon.case` – Case Components

Defines the Pylon power system model.

class `Case`(*name=None, base_mva=100.0, buses=None, branches=None, generators=None*)
Bases: `pylon.util.Named`, `pylon.util.Serializable`

Defines representation of an electric power system as a graph of Bus objects connected by Branches.

`Bdc`
Returns the sparse susceptance matrices and phase shift injection vectors needed for a DC power flow [2].

The bus real power injections are related to bus voltage angles by P = Bbus * Va + Pbusinj

The real power flows at the from end the lines are related to the bus voltage angles by

Pf = Bf * Va + Pfinj

Pf | | Bff Bft | | Vaf | | Pfinj |
| = | | * | | + | | Pt | | Btf Btt | | Vat | | Ptinj |

[2] Ray Zimmerman, "makeBdc.m", MATPOWER, PSERC Cornell,
http://www.pserc.cornell.edu/matpower/, version 4.0b1, Dec 2009

`Sbus`
Net complex bus power injection vector in p.u.

`Y`
Returns the bus and branch admittance matrices, Yf and Yt, such that Yf * V is the vector of complex branch currents injected at each branch's "from" bus [1].

[1] Ray Zimmerman, "makeYbus.m", MATPOWER, PSERC Cornell,
http://www.pserc.cornell.edu/matpower/, version 4.0b1, Dec 2009

`connected_buses`
Returns a list of buses that are connected to one or more branches or the first bus in a branchless system.

`d2AIbr_dV2`(*dIbr_dVa, dIbr_dVm, Ibr, Ybr, V, lam*)
Computes 2nd derivatives of |complex current|**2 w.r.t. V.

`d2ASbr_dV2`(*dSbr_dVa, dSbr_dVm, Sbr, Cbr, Ybr, V, lam*)
Computes 2nd derivatives of |complex power flow|**2 w.r.t. V.

`d2Ibr_dV2`(*Ybr, V, lam*)
Computes 2nd derivatives of complex branch current w.r.t. voltage.

d2Sbr_dV2(*Cbr, Ybr, V, lam*)
Computes 2nd derivatives of complex power flow w.r.t. voltage.

d2Sbus_dV2(*Ybus, V, lam*)
Computes 2nd derivatives of power injection w.r.t. voltage.

dAbr_dV(*dSf_dVa, dSf_dVm, dSt_dVa, dSt_dVm, Sf, St*)
Partial derivatives of squared flow magnitudes w.r.t voltage.

Computes partial derivatives of apparent power w.r.t active and reactive power flows. Partial derivative must equal 1 for lines with zero flow to avoid division by zero errors (1 comes from L'Hopital).

dIbr_dV(*Yf, Yt, V*)
Computes partial derivatives of branch currents w.r.t. voltage [4].

[4] Ray Zimmerman, "dIbr_dV.m", MATPOWER, version 4.0b1, PSERC (Cornell), http://www.pserc.cornell.edu/matpower/

dSbr_dV(*Yf, Yt, V, buses=None, branches=None*)
Computes the branch power flow vector and the partial derivative of branch power flow w.r.t voltage.

dSbus_dV(*Y, V*)
Computes the partial derivative of power injection w.r.t. voltage [3].

[3] Ray Zimmerman, "dSbus_dV.m", MATPOWER, version 4.0b1, PSERC (Cornell), http://www.pserc.cornell.edu/matpower/

deactivate_isolated()
Deactivates branches and generators connected to isolated buses.

getSbus(*buses=None*)
Net complex bus power injection vector in p.u.

getYbus(*buses=None, branches=None*)
Returns the bus and branch admittance matrices, Yf and Yt, such that Yf * V is the vector of complex branch currents injected at each branch's "from" bus [1].

[1] Ray Zimmerman, "makeYbus.m", MATPOWER, PSERC Cornell, http://www.pserc.cornell.edu/matpower/, version 4.0b1, Dec 2009

index_branches(*branches=None*)
Updates the indices for all brnaches.

index_buses(*buses=None*)
Updates the indices of all case buses.

class **load_matpower**(*fd*)
Returns a case from the given MATPOWER file object.

class **load_psat**(*fd*)
Returns a case object from the given PSAT data file.

class **load_psse**(*fd*)
Returns a case from the given PSS/E file object.

makeB(*buses=None, branches=None, method='XB'*)
Builds the FDPF matrices, B prime and B double prime.

makeBdc(*buses=None, branches=None*)
Returns the sparse susceptance matrices and phase shift injection vectors needed for a DC power flow [2].

The bus real power injections are related to bus voltage angles by P = Bbus * Va + Pbusinj

The real power flows at the from end the lines are related to the bus voltage angles by

Pf = Bf * Va + Pfinj

```
Pf |   | Bff  Bft |   | Vaf |   | Pfinj |
| = |         | * |     | + |       | Pt | | Btf Btt | | Vat | | Ptinj |
```

[2] Ray Zimmerman, "makeBdc.m", MATPOWER, PSERC Cornell,
http://www.pserc.cornell.edu/matpower/, version 4.0b1, Dec 2009

online_branches
Property getter for in-service branches.

online_generators
All in-service generators.

pf_solution(*Ybus, Yf, Yt, V*)
Updates buses, generators and branches to match power flow solution.

reset()
Resets the result variables for all of the case componenets.

s_demand(*bus*)
Returns the total complex power demand.

s_supply(*bus*)
Returns the total complex power generation capacity.

s_surplus(*bus*)
Return the difference between supply and demand.

save_csv(*fd*)
Saves the case as a series of Comma-Separated Values.

save_dot(*fd*)
Saves a representation of the case in the Graphviz DOT language.

save_excel(*fd*)
Saves the case as an Excel spreadsheet.

save_matpower(*fd*)
Serialize the case as a MATPOWER data file.

save_rst(*fd*)
Save a reStructuredText representation of the case.

sort_generators()
Reorders the list of generators according to bus index.

class Bus(*name=None, type='PQ', v_base=100.0, v_magnitude_guess=1.0, v_angle_guess=0.0, v_max=1.1000000000000001, v_min=0.90000000000000002, p_demand=0.0, q_demand=0.0, g_shunt=0.0, b_shunt=0.0, position=None*)
Bases: pylon.util.Named

Defines a power system bus node.

reset()
Resets the result variables.

class Branch(*from_bus, to_bus, name=None, online=True, r=0.0, x=0.0, b=0.0, rate_a=999.0, rate_b=999.0, rate_c=999.0, ratio=1.0, phase_shift=0.0, ang_min=-360.0, ang_max=360.0*)
Bases: pylon.util.Named

Defines a case edge that links two Bus objects.

reset()
Resets the result variables.

Defines a generator as a complex power bus injection.

class `Generator`(*bus, name=None, online=True, base_mva=100.0, p=100.0, p_max=200.0, p_min=0.0, v_magnitude=1.0, q=0.0, q_max=30.0, q_min=-30.0, c_startup=0.0, c_shutdown=0.0, p_cost=None, pcost_model='poly', q_cost=None, qcost_model=None*)

Bases: `pylon.util.Named`

Defines a power system generator component. Fixes voltage magnitude and active power injected at parent bus. Or when at it's reactive power limit fixes active and reactive power injected at parent bus.

`bids_to_pwl`(*bids*)

Updates the piece-wise linear total cost function using the given bid blocks.

@see: matpower3.2/extras/smartmarket/off2case.m

`get_bids`(*n_points=6*)

Returns quantity and price bids created from the cost function.

`get_offers`(*n_points=6*)

Returns quantity and price offers created from the cost function.

`is_load`

Returns true if the generator if a dispatchable load. This may need to be revised to allow sensible specification of both elastic demand and pumped storage units.

`offers_to_pwl`(*offers*)

Updates the piece-wise linear total cost function using the given offer blocks.

@see: matpower3.2/extras/smartmarket/off2case.m

`poly_to_pwl`(*n_points=10*)

Sets the piece-wise linear cost attribute, converting the polynomial cost variable by evaluating at zero and then at n_points evenly spaced points between p_min and p_max.

`pwl_to_poly`()

Converts the first segment of the pwl cost to linear quadratic. FIXME: Curve-fit for all segments.

`q_limited`

Is the machine at it's limit of reactive power?

`reset`()

Resets the result variables.

`total_cost`(*p=None, p_cost=None, pcost_model=None*)

Computes total cost for the generator at the given output level.

4.2 `pylon.dc_pf` – DC Power Flow

Defines a solver for DC power flow [1].

[1] Ray Zimmerman, "dcpf.m", MATPOWER, PSERC Cornell, version 3.2, http://www.pserc.cornell.edu/matpower/, June 2007

class `DCPF`(*case, solver='UMFPACK'*)

Bases: `object`

Solves DC power flow [1].

[1] Ray Zimmerman, "dcpf.m", MATPOWER, PSERC Cornell, version 3.2, http://www.pserc.cornell.edu/matpower/, June 2007

solve()
Solves DC power flow for the given case.

4.3 pylon.ac_pf – AC Power Flow

Defines solvers for AC power flow [1].

[1] Ray Zimmerman, "runpf.m", MATPOWER, PSERC Cornell, http://www.pserc.cornell.edu/matpower/, version 4.0b1, Dec 2009

class _ACPF(*case, qlimit=False, tolerance=1e-08, iter_max=10, verbose=True*)
Bases: object

Defines a base class for AC power flow solvers [1].

[1] Ray Zimmerman, "runpf.m", MATPOWER, PSERC Cornell, http://www.pserc.cornell.edu/matpower/, version 4.0b1, Dec 2009

solve()
Override this method in subclasses.

class NewtonPF(*case, qlimit=False, tolerance=1e-08, iter_max=10, verbose=True*)
Bases: pylon.ac_pf._ACPF

Solves the power flow using full Newton's method [2].

[2] Ray Zimmerman, "newtonpf.m", MATPOWER, PSERC Cornell, http://www.pserc.cornell.edu/matpower/, version 4.0b1, Dec 2009

class FastDecoupledPF(*case, qlimit=False, tolerance=1e-08, iter_max=20, verbose=True, method='XB'*)
Bases: pylon.ac_pf._ACPF

Solves the power flow using fast decoupled method [3].

[3] Ray Zimmerman, "fdpf.m", MATPOWER, PSERC Cornell, version 4.0b1, http://www.pserc.cornell.edu/matpower/, December 2009

4.4 pylon.opf – Optimal Power Flow

Defines a generalised OPF solver and an OPF model [1].

[1] Ray Zimmerman, "opf.m", MATPOWER, PSERC Cornell, version 4.0b1, http://www.pserc.cornell.edu/matpower/, December 2009

class OPF(*case, dc=True, ignore_ang_lim=True, opt=None*)
Bases: object

Defines a generalised OPF solver [1].

[1] Ray Zimmerman, "opf.m", MATPOWER, PSERC Cornell, version 4.0b1,
http://www.pserc.cornell.edu/matpower/, December 2009

solve(*solver_klass=None*)
Solves an optimal power flow and returns a results dictionary.

4.5 Indices and tables

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

P

INDEX

T

Y