# CONTENTS

1 Introduction 1

2 Installation 3
   2.1 Dependencies ....................................................... 3
   2.2 Recommended ..................................................... 3
   2.3 Setup tools ......................................................... 3
   2.4 Installation from source ......................................... 4
   2.5 Working directory ............................................... 4

3 Tutorial 5
   3.1 Power Flow ......................................................... 5
   3.2 Optimal Power Flow .............................................. 6

4 API 7
   4.1 pylon.case – Case Components .................................. 7
   4.2 pylon.dc_pf – DC Power Flow .................................... 10
   4.3 pylon.ac_pf – AC Power Flow .................................... 11
   4.4 pylon.opf – Optimal Power Flow ............................... 11
   4.5 Indices and tables ................................................ 12

Module Index 13

Index 15
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

scikit.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-0.4.1
$ ipython

Access the `pylon` application programming interface.

In [1]: from pylon import Case, OPF
3.1 Power Flow

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

```python
from pylon import Case, Bus, Branch, Generator, NewtonPF, FastDecoupledPF
```

Import “sys” so the report can be written to stdout.

```python
import sys
```

Start by building up a one branch case with a generator at one end

```python
bus1 = Bus()
g = Generator(bus1, p=80.0, q=10.0)
```

and fixed load at the other.

```python
bus2 = Bus(p_demand=60.0, q_demand=4.0)
```

Connect the two buses

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

and add it all to a new case.

```python
case = Case(buses=[bus1, bus2], branches=[line], generators=[g])
```

Choose to solve using either Newton’s method

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

or Fast Decoupled method

```python
solver = FastDecoupledPF(case).solve()
```

and then call the solver.

```python
solver.solve()
```

Write the case out to view the results.
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.

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

Import “sys” for writing to stdout.

```python
import sys
```

Create two generators, specifying their marginal cost.

```python
bus1 = Bus(p_demand=100.0)
g1 = Generator(bus1, p_min=0.0, p_max=80.0, p_cost=(0.0, 6.0, 0.0))
bus2 = Bus()
g2 = Generator(bus2, p_min=0.0, p_max=60.0, p_cost=(0.0, 9.0, 0.0))
```

Connect the two generator buses

```python
line = Branch(bus1, bus2, r=0.05)
```

and add it all to a case.

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

Linearised DC optimal power flow

```python
dc = True
```

or non-linear AC optimal power flow may be selected.

```python
dc = False
```

Pass the case to the OPF routine and solve.

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

View the results as ReStructuredText.

```python
case.save_rst(sys.stdout)
```
4.1 pylon.case – Case Components

Defines the Pylon power system model.

```python
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**

\[
P = \text{Bbus} \cdot \text{Va} + \text{Pbusinj}
\]

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

\[
P_f = \text{Bf} \cdot \text{Va} + \text{Pfinj}
\]

\[
\begin{bmatrix}
P_f \\
P_t
\end{bmatrix} =
\begin{bmatrix}
\text{Bff} & \text{Bft} \\
\text{Btf} & \text{Btt}
\end{bmatrix}
\begin{bmatrix}
\text{Vaf} \\
\text{Vat}
\end{bmatrix} +
\begin{bmatrix}
P_{finj} \\
P_{tinj}
\end{bmatrix}
\]


**Sbus**

Net complex bus power injection vector in p.u.

**Y**

Returns the bus and branch admittance matrices, \(Y_f\) and \(Y_t\), such that \(Y_f \cdot V\) is the vector of complex branch currents injected at each branch’s “from” bus [1].


**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**

\(dI_{br} \cdot dVa, dI_{br} \cdot dVm, I_{br}, Y_{br}, V, \text{lam}\)

Computes 2nd derivatives of complex current\(^2\) w.r.t. V.

**d2ASbr_dV2**

\(dS_{br} \cdot dVa, dS_{br} \cdot dVm, S_{br}, C_{br}, Y_{br}, V, \text{lam}\)

Computes 2nd derivatives of complex power flow\(^2\) w.r.t. V.

**d2Ibr_dV2**

\(Y_{br}, V, \text{lam}\)

Computes 2nd derivatives of complex branch current w.r.t. voltage.
\textit{d2Sbr\textsubscript{d} dV\textsuperscript{2}} (Cbr, Ybr, V, lam)
Computes 2nd derivatives of complex power flow w.r.t. voltage.

\textit{d2Sbus\textsubscript{d} dV\textsuperscript{2}} (Ybus, V, lam)
Computes 2nd derivatives of power injection w.r.t. voltage.

\textit{dAbr\textsubscript{d} dV} (dSf\textsubscript{d Va}, dSf\textsubscript{d Vm}, dSt\textsubscript{d Va}, dSt\textsubscript{d Vm}, 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).

\textit{dIbr\textsubscript{d} dV} (Yf, Yt, V)
Computes partial derivatives of branch currents w.r.t. voltage [4].

\textit{dSbr\textsubscript{d} 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.

\textit{dSbus\textsubscript{d} dV} (Y, V)
Computes the partial derivative of power injection w.r.t. voltage [3].

\textit{deactivate\_isolated}()
Deactivates branches and generators connected to isolated buses.

\textit{getSbus} (buses=None)
Net complex bus power injection vector in p.u.

\textit{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].

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

\textit{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 \( P = Bbus \ast Va + Pbusinj \)
The real power flows at the from end the lines are related to the bus voltage angles by
\[ Pf = Bf \cdot Va + P_{\text{finj}} \]

<table>
<thead>
<tr>
<th>Pf</th>
<th>Bff</th>
<th>Bft</th>
<th>Vaf</th>
<th>P_{\text{finj}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>*</td>
<td>+</td>
<td></td>
<td>Pt</td>
</tr>
</tbody>
</table>


**online_branches**
Property getter for in-service branches.

**online_generators**
All in-service generators.

**pf_solution** (\(Y_{bus}, Y_{f}, Y_{t}, V\))
Updates buses, generators and branches to match power flow solution.

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

**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)
Serializes 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=0.9000000000000002, 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.

```python
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.

```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.

```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].


```class DCPF(case, solver='UMFPACK')
    Bases: object

Solves DC power flow [1].

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

4.3 pylon.ac_pf – AC Power Flow

Defines solvers for AC power flow [1].


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].
	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].

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].

4.4 pylon.opf – Optimal Power Flow

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


class OPF (case, dc=True, ignore_ang_lim=True, opt=None)
    Bases: object
    Defines a generalised OPF solver [1].
	solve (solver_klass=None)
    Solves an optimal power flow and returns a results dictionary.
4.5 Indices and tables

- Index
- Module Index
- Search Page
P
  pylon.ac_pf, 11
  pylon.case, 7
  pylon.dc_pf, 10
  pylon.generator, 10
  pylon.opf, 11
Symbols

_ACPF (class in pylon.ac_pf), 11
B
Bdc (pylon.case.Case attribute), 7
bids_to_pwl() (pylon.generator.Generator method), 10
Branch (class in pylon.case), 9
Bus (class in pylon.case), 9
C
Case (class in pylon.case), 7
connected_buses (pylon.case.Case attribute), 7
D
d2Albr_dV2() (pylon.case.Case method), 7
d2ASbr_dV2() (pylon.case.Case method), 7
d2lbr_dV2() (pylon.case.Case method), 7
d2Sbr_dV2() (pylon.case.Case method), 7
d2Sbus_dV2() (pylon.case.Case method), 8
dAbr_dV() (pylon.case.Case method), 8
DCPF (class in pylon.dc_pf), 10
deactivate_isolated() (pylon.case.Case method), 8
dlbr_dV() (pylon.case.Case method), 8
dSbr_dV() (pylon.case.Case method), 8
dSbus_dV() (pylon.case.Case method), 8
F
FastDecoupledPF (class in pylon.ac_pf), 11
G
Generator (class in pylon.generator), 10
going_to_bids() (pylon.generator.Generator method), 10
going_to_offers() (pylon.generator.Generator method), 10
going_to_Sbus() (pylon.case.Case method), 8
getYbus() (pylon.case.Case method), 8
I
index_branches() (pylon.case.Case method), 8
index_buses() (pylon.case.Case method), 8
is_load (pylon.generator.Generator attribute), 10
L
load_matpower() (pylon.case.Case class method), 8
load_psat() (pylon.case.Case class method), 8
load_psse() (pylon.case.Case class method), 8
M
makeB() (pylon.case.Case method), 8
makeBdc() (pylon.case.Case method), 8
N
NewtonPF (class in pylon.ac_pf), 11
O
offers_to_pwl() (pylon.generator.Generator method), 10
online_branches (pylon.case.Case attribute), 9
online_generators (pylon.case.Case attribute), 9
OPF (class in pylon.opf), 11
P
pf_solution() (pylon.case.Case method), 9
poly_to_pwl() (pylon.generator.Generator method), 10
pwl_to_poly() (pylon.generator.Generator method), 10
pylon.ac_pf (module), 11
pylon.case (module), 7
pylon.dc_pf (module), 10
pylon.generator (module), 10
pylon.opf (module), 11
Q
q_limited (pylon.generator.Generator attribute), 10
R
reset() (pylon.case.Branch method), 9
reset() (pylon.case.Bus method), 9
reset() (pylon.case.Case method), 9
reset() (pylon.generator.Generator method), 10
S
s_demand() (pylon.case.Case method), 9
s_supply() (pylon.case.Case method), 9
s_surplus() (pylon.case.Case method), 9
save_csv() (pylon.case.Case method), 9
save_dot() (pylon.case.Case method), 9
save_excel() (pylon.case.Case method), 9
save_matpower() (pylon.case.Case method), 9
save_rst() (pylon.case.Case method), 9
Sbus (pylon.case.Case attribute), 7
solve() (pylon.ac_pf.ACPF method), 11
solve() (pylon.dc_pf.DCPF method), 10
solve() (pylon.opf.OPF method), 11
sort_generators() (pylon.case.Case method), 9

t

T

total_cost() (pylon.generator.Generator method), 10

Y

Y (pylon.case.Case attribute), 7