
GRIDOPT Documentation

Release 1.0

Tomas Tinoco De Rubira

March 11, 2016

CONTENTS

1	Getting Started	3
1.1	Download	3
1.2	Installation	3
1.3	Docker	3
1.4	Example	4
2	Power Flow	7
2.1	DCPF	8
2.2	DCOPF	8
2.3	NRPF	9
2.4	AugLPF	9
2.5	AugLOPF	9
3	API Reference	11
3.1	Power Flow Method	11
3.2	References	13
4	Indices and tables	15
	Bibliography	17
	Python Module Index	19
	Python Module Index	21
	Index	23

Welcome! This is the documentation for GRIDOPT, last updated March 11, 2016.

What is GRIDOPT?

GRIDOPT is a Python package that provides methods for solving power grid optimization problems.

License

GRIDOPT is released under the BSD 2-clause license.

Citing

If you use GRIDOPT in your work, please cite the software as follows:

```
@misc{gridopt,
  author={Tinoco De Rubira, Tomas},
  title={{GRIDOPT}: {A} {P}ython package for power grid optimization},
  howpublished={\url{https://github.com/ttinoco/GRIDOPT}},
  month={March},
  year={2016}
}
```

Contact

If you have any questions about GRIDOPT or if you are interested in collaborating, send me an email:

- Tomas Tinoco De Rubira (ttinoco5687@gmail.com).

Documentation Contents

GETTING STARTED

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

- [Numpy](#) ($\geq 1.8.2$)
- [Scipy](#) ($\geq 0.13.3$)
- [OPTALG](#)
- [PFNET](#) (≥ 1.1)

1.1 Download

The latest version of GRIDOPT can be downloaded from <https://github.com/ttinoco/GRIDOPT>.

1.2 Installation

The GRIDOPT Python module can be installed using:

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

from the root directory of the package.

The installation can be tested using [nose](#) as follows:

```
> nosetests -v
```

1.3 Docker

If GRIDOPT was obtained as a [Docker](#) image, say [gridopt.tar](#), then one needs to first install [Docker Engine](#). Then one can load the image using the command:

```
> docker load -i gridopt.tar
```

and enter the application environment with:

```
> docker run -i -t --entrypoint=/bin/bash gridopt
```

In the application environment, GRIDOPT and all its dependencies, *e.g.*, [PFNET](#), are already installed and ready to go. There, one can navigate to the directory `/gridopt/tests/resources` and use the test cases available there to do the [PFNET](#) and GRIDOPT tutorials with Python.

1.3.1 Graphics in Linux

To display graphics within the application environment, the following command can be run for entering the application environment:

```
> docker run -i -t --entrypoint=/bin/bash -e DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix:ro gridopt
```

Then, on the host machine the command `xhost +` can be used to enable access to your host machine's display and then `xhost -` to disable it after usage. Inside the application environment, the command `xeyes` can be used to check whether graphics are working.

1.3.2 Graphics in Windows

Displaying graphics on Windows involves a few more steps. First, [Xming](#), an X server for Windows, must be downloaded and installed. Then, the installed application XLaunch should be executed with the options `Multiple windows`, `Display number 0`, `Start no client`, `Clipboard` and `No Access Control`. Once this is done, the application environment can be entered using:

```
> docker run -i -t --entrypoint=/bin/bash -e DISPLAY=ip_address_taken_by_Xming:0 gridopt
```

where the IP address taken by Xming can be found in the file `C:\Users\username\AppData\Local\Temp\Xming.0` next to `XdmcpRegisterConnection`. Again, graphics within the application environment can be tested using the command `xeyes`.

1.3.3 Graphics in Mac

Coming soon.

1.4 Example

The next example shows how to solve the power flow problem associated with a power grid using GRIDOPT:

```
>>> import pfnet
>>> import gridopt

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

>>> # max mismatches (MW, MVar)
>>> print '%.2e %.2e' %(net.bus_P_mis, net.bus_Q_mis)
3.54e-01 4.22e+00

>>> method = gridopt.power_flow.new_method('NRPF')

>>> method.set_parameters({'quiet': True})

>>> method.solve(net)

>>> results = method.get_results()

>>> print results['status']
solved

>>> method.update_network(net)
```



```
>>> # max mismatches (MW,MVar)
>>> print '%.2e %.2e' %(net.bus_P_mis,net.bus_Q_mis)
5.16e-04 5.67e-03
```


POWER FLOW

The Power Flow (PF) problem consists of determining steady-state voltage magnitudes and angles at every bus of the network as well as any unknown generator powers. On the other hand, the Optimal Power Flow (OPF) problem consists of determining generator and other network control settings that result in the optimal operation of the network according to some measure, *e.g.*, generation cost. In GRIDOPT, methods for solving PF and OPF problems are represented by objects derived from a `method` base class.

To solve a PF or OPF problem, one first needs to create an instance of a specific method subclass. This is done using the function `new_method()`, which takes as argument the name of an available power flow method (*DCPF*, *DCOPF*, *NRPF*, *AugLPF*, and *AugLOPF*). The following code sample creates an instance of the *NRPF* method:

```
>>> import gridopt

>>> method = gridopt.power_flow.new_method('NRPF')
```

Once a method has been instantiated, its parameters can be set using the function `set_parameters()`. This function takes as argument a dictionary with parameter name and value pairs. Valid parameters include parameters of the method, which are described in the sections below, and parameters from the numerical solver used by the method. The numerical solvers used by the methods of GRIDOPT belong to the Python package *OPTALG*. The following code sample sets a few parameters of the method created above:

```
>>> method.set_parameters({'quiet': True, 'feastol': 1e-4})
```

After configuring parameters, a method can be used to solve a problem using the function `solve()`. This function takes as argument a *PFNET* Network object, as follows:

```
>>> import pfnet

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

>>> method.solve(net)
```

Information about the execution of the method can be obtained from the `results` attribute of the `method` object. This dictionary of results includes information such as 'status', *e.g.*, 'solved' or 'error', any error message ('error_msg'), solver 'iterations', the *PFNET* optimization 'problem' constructed, and the properties of the *PFNET* Network at the point found by the method ('net_properties'). The following code sample shows how to extract some results:

```
>>> results = method.get_results()

>>> print results['status']
solved

>>> print results['iterations']
1
```

```
>>> problem = results['problem']
>>> problem.show()

Problem
functions : 0
constraints: 4
  type: FIX
  type: PAR_GEN_Q
  type: PAR_GEN_P
  type: PF

>>> print results['net_properties']['bus_v_max']
1.09
```

If desired, one can update the **PFNET** Network object with the solution found by the method. This can be done with the function `update_network()`. This routine not only updates the network quantities treated as variables by the method, but also information about the sensitivity of the optimal objective function value with respect to perturbations of the constraints. The following code sample updates the power network with the results obtained by the method and shows the resulting maximum active and reactive bus power mismatches in units of MW and MVar:

```
>>> method.update_network(net)

>>> print '%.2e %.2e' %(net.bus_P_mis, net.bus_Q_mis)
5.16e-04 5.67e-03
```

2.1 DCPF

This method solves a DC power flow problem, which is just a linear system of equations. For doing this, it uses one of the linear solvers from **OPTALG**.

2.2 DCOPF

This method solves a DC optimal power flow problem, which is just a quadratic program including active power generation cost, power balance, generator limits, and branch thermal limits. For doing this, it uses the `OptSolverIQP` interior point solver from **OPTALG**. For now, its only parameter is a `'quiet'` flag.

The following example illustrates how to solve a DCOPF problem and extract the optimal generation cost:

```
>>> method = gridopt.power_flow.new_method('DCOPF')

>>> method.solve(net)

>>> print method.results['status']
solved

>>> method.update_network(net)

>>> # generation cost ($/hour)
>>> print net.gen_P_cost
5545.10
```

The sensitivity of the optimal generation cost with respect to the power balance equations can be easily extracted from the network buses:

```
>>> bus = net.get_bus(4)
>>> print "bus %2d %.2e" %(bus.index, bus.sens_P_balance)
bus 4 2.13e+01
```

Similarly, the sensitivity with respect to branch flow limits can be easily extracted from the network branches:

```
>>> branch = net.get_branch(6)
>>> print "branch %2d %.2e %.2e" %(branch.index,
...                               branch.sens_P_u_bound,
...                               branch.sens_P_l_bound)
branch 6 2.01e-09 1.25e-09
```

Lastly, the sensitivity with respect to generator active power limits can be easily extracted from the network generators:

```
>>> gen = net.get_gen(2)
>>> print "gen %2d %.2e %.2e" %(gen.index,
...                              gen.sens_P_u_bound,
...                              gen.sens_P_l_bound)
gen 2 2.01e-06 2.85e+01
```

As the examples show, GRIDOPT and PFNET take care of all the details and allow one to extract solution information easily and intuitively from the network components.

2.3 NRPF

This method solves an AC power flow problem, which is a nonlinear system of equations. For doing this, it uses the `OptSolverNR` Newton-Raphson solver from [OPTALG](#). For now, its parameters are a `'quiet'` flag and a low-voltage threshold `'vmin_thresh'`.

2.4 AugLPF

This method solves an AC power flow problem but formulated as an optimization problem with a strongly-convex objective function. For doing this, it uses the `OptSolverAugL` Augmented Lagrangian solver from [OPTALG](#). The `OptSolverAugL` solver is similar to the one described in Chapter 3 of [\[TTR2015\]](#), but without the restriction of moving in the null-space of the linear equality constraints. For now, the parameters of this power flow method are the following:

Name	Description	Default
<code>'weight_vmag'</code>	Weight for bus voltage magnitude regularization	1e0
<code>'weight_vang'</code>	Weight for bus voltage angle regularization	1e-3
<code>'weight_pq'</code>	Weight for generator power regularization	1e-3
<code>'weight_t'</code>	Weight for transformer tap ratio regularization	1e1
<code>'weight_b'</code>	Weight for shunt susceptance regularization	1e-4
<code>'vmin_thresh'</code>	Low-voltage threshold	1e-1

2.5 AugLOPF

This method solves an AC optimal power flow problem. For doing this, it uses the `OptSolverAugL` Augmented Lagrangian solver from [OPTALG](#). For now, the parameters of this optimal power flow method are the following:

Name	Description	Default
'weight_cost'	Weight for active power generation cost	1e-2
'weight_limit'	Weight for soft constraint violations	1e-2
'weight_reg'	Weight for regularization	1e-5
'vmin_thresh'	Low-voltage threshold	1e-1

API REFERENCE

3.1 Power Flow Method

`gridopt.power_flow.new_method(name)`

Creates a power flow or optimal power flow method.

Parameters `name` : string

class `gridopt.power_flow.method.PFmethod`

Power flow method class.

create_problem (*net*)

Creates optimization problem.

Parameters `net` : PFNET Network

Returns `prob` : PFNET Problem

get_info_printer ()

Gets function for printing information about method progress.

Returns `printer` : Function

get_results ()

Gets dictionary with results.

Returns `results` : dict

parameters = None

Parameters (dictionary)

results = None

Results (dictionary)

set_dual_variables (*d*)

Sets dual variables.

Parameters `d` : list

set_error_msg (*msg*)

Sets method error message.

Parameters `msg` : string

set_iterations (*k*)

Sets method iterations.

Parameters `k` : int

set_net_properties (*np*)

Sets network properties.

Parameters *np* : dictionary

set_parameters (*params=None, strparams=None*)

Sets method parameters.

Parameters *params* : dict

Name-value pairs

strparams: dict :

Name-value pairs where value is a string

set_primal_variables (*x*)

Sets primal variables.

Parameters *x* : vector

set_problem (*p*)

Sets problem.

Parameters *p* : PFNET problem

set_results (*results*)

Sets method results.

Parameters *results* : dict

set_status (*status*)

Sets method status.

Parameters *status* : string

solve (*net*)

Solves power flow problem.

Parameters *net* : PFNET Network

update_network (*net*)

Updates network with results.

Parameters *net* : PFNET Network

class `gridopt.power_flow.DCPF`

DC power flow method.

class `gridopt.power_flow.DCOPF`

DC optimal power flow method.

class `gridopt.power_flow.NRPF`

Newton-Raphson power flow method.

class `gridopt.power_flow.AugLPF`

Augmented Lagrangian-based power flow method.

class `gridopt.power_flow.AugLOPF`

Augmented Lagrangian-based optimal power flow method.

class `gridopt.power_flow.method_error.PFmethodError` (*method, msg*)

3.1.1 Error Exceptions

```
class gridopt.power_flow.method_error.PFmethodError(method, msg)
class gridopt.power_flow.method_error.PFmethodError_NoProblem(method)
class gridopt.power_flow.method_error.PFmethodError_BadProblem(method)
class gridopt.power_flow.method_error.PFmethodError_BadFlowLimits(method)
class gridopt.power_flow.method_error.PFmethodError_BadVarLimits(method)
class gridopt.power_flow.method_error.PFmethodError_BadParam(method, param='')
class gridopt.power_flow.method_error.PFmethodError_ParamNotBool(method)
class gridopt.power_flow.method_error.PFmethodError_SolverError(method, msg)
```

3.2 References

INDICES AND TABLES

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

BIBLIOGRAPHY

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

g

gridopt, 1

g

`gridopt`, 1

A

AugLOPF (class in `gridopt.power_flow`), 12
AugLPF (class in `gridopt.power_flow`), 12

C

`create_problem()` (`gridopt.power_flow.method.PFmethod` method), 11

D

DCOPF (class in `gridopt.power_flow`), 12
DCPF (class in `gridopt.power_flow`), 12

G

`get_info_printer()` (`gridopt.power_flow.method.PFmethod` method), 11
`get_results()` (`gridopt.power_flow.method.PFmethod` method), 11
`gridopt` (module), 1

N

`new_method()` (in module `gridopt.power_flow`), 11
NRPF (class in `gridopt.power_flow`), 12

P

`parameters` (`gridopt.power_flow.method.PFmethod` attribute), 11
`PFmethod` (class in `gridopt.power_flow.method`), 11
`PFmethodError` (class in `gridopt.power_flow.method_error`), 12, 13
`PFmethodError_BadFlowLimits` (class in `gridopt.power_flow.method_error`), 13
`PFmethodError_BadParam` (class in `gridopt.power_flow.method_error`), 13
`PFmethodError_BadProblem` (class in `gridopt.power_flow.method_error`), 13
`PFmethodError_BadVarLimits` (class in `gridopt.power_flow.method_error`), 13
`PFmethodError_NoProblem` (class in `gridopt.power_flow.method_error`), 13
`PFmethodError_ParamNotBool` (class in `gridopt.power_flow.method_error`), 13

`PFmethodError_SolverError` (class in `gridopt.power_flow.method_error`), 13

R

`results` (`gridopt.power_flow.method.PFmethod` attribute), 11

S

`set_dual_variables()` (`gridopt.power_flow.method.PFmethod` method), 11
`set_error_msg()` (`gridopt.power_flow.method.PFmethod` method), 11
`set_iterations()` (`gridopt.power_flow.method.PFmethod` method), 11
`set_net_properties()` (`gridopt.power_flow.method.PFmethod` method), 11
`set_parameters()` (`gridopt.power_flow.method.PFmethod` method), 12
`set_primal_variables()` (`gridopt.power_flow.method.PFmethod` method), 12
`set_problem()` (`gridopt.power_flow.method.PFmethod` method), 12
`set_results()` (`gridopt.power_flow.method.PFmethod` method), 12
`set_status()` (`gridopt.power_flow.method.PFmethod` method), 12
`solve()` (`gridopt.power_flow.method.PFmethod` method), 12

U

`update_network()` (`gridopt.power_flow.method.PFmethod` method), 12