GRIDOPT Documentation

Release 1.3.7

Tomas Tinoco De Rubira

Jan 07, 2020

Contents

Welcome! This is the documentation for GRIDOPT version 1.3.7, last updated Jan 07, 2020.

What is GRIDOPT?

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

License

Contents

CHAPTER 1

Getting Started

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

1.1 Installation

In order to install GRIDOPT, the following tools are needed:

- Linux and macOS:
  - C compiler
  - Make
  - Python (2 or 3)
  - pip
- Windows:
  - Anaconda (for Python 2.7)
  - CMake (choose "Add CMake to the system PATH for all users" during installation)
  - 7-Zip (update system path to include the 7z executable, typically in `C:\Program Files\7-Zip`)
  - mingwpy (use `pip install -i https://pypi.anaconda.org/carlkl/simple mingwpy`)

After getting these tools, the GRIDOPT Python module can be easily installed by executing the following commands on the terminal or Anaconda prompt:

```
pip install numpy cython
pip install optalg pfnet
pip install gridopt
```

To install the module from source, the code can be obtained from https://github.com/ttinoco/GRIDOPT, and then the following commands can be executed on the terminal or Anaconda prompt from the root directory of the package:

```
pip install numpy cython
pip install optalg pfnet
python setup.py install
```

Running the unit tests can be done with:

```
pip install nose
nosetests -s -v
```

1.2 Example

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

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

>>> net = pfnet.PyParserMAT().parse('ieee14.m')

>>> # 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('ACPF')

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

>>> method.solve(net)

>>> results = method.get_results()

>>> print results['solver status']
solved

>>> method.update_network(net)

>>> # max mismatches (MW,MVAr)
>>> print '%.2e %.2e' %(net.bus_P_mis,net.bus_Q_mis)
5.09e-06 9.70e-06
```

In this example, it is assumed that the Python interpreter is started from a directory where the sample case ieee14 is located.

CHAPTER 2

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 powers and 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 PF or OPF method. Available methods are the following:

- *DCPF*
- *DCOPF*
- *ACPF*
- *ACOPF*

The following code sample creates an instance of the *ACPF* method:

```
>>> import gridopt

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

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-value pairs. Valid parameters include parameters of the method, which are described in the sections below, and parameters of 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({'solver': 'nr', '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.PyParserMAT().parse('ieee14.m')

>>> 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 `'solver status'`, *e.g.*, `'solved'` or `'error'`, any `'solver message'`, `'solver iterations'`, a `'network snapshot'` reflecting the solution, and others. The following code sample shows how to extract some results:

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

>>> print results['solver status']
solved

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

>>> print results['network snapshot'].bus_v_max
1.09
```

If desired, one can update the input `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 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)
5.09e-06 9.70e-06
```

2.1 DCPF

This method is represented by an object of type *DCPF* and solves a DC power flow problem, which is just a linear system of equations representing DC Power balance constraints.

The parameters of this method are the following:

Name	Description	Default
`'solver'`	OPTALG linear solver `{'superlu','mumps','umfpack'}`	`'superlu'`
`'quiet'`	Flag for disabling output	`False`

2.2 DCOPF

This method is represented by an object of type *DCOPF* and solves a DC optimal power flow problem, which is just a quadratic program that considers Active power generation cost, Active power consumption utility, DC Power balance, Variable bounds, *e.g.*, generator and load limits, and DC branch flow limits.

The parameters of this method are the following:

Name	Description	Default
`'thermal_limits'`	Flag for considering branch flow limits	`False`
`'renewable_curtailment'`	Flag for allowing curtailment of renewable generators	`False`
`'solver'`	OPTALG optimization solver `{'iqp','inlp','augl','ipopt'}`	`'iqp'`

2.3 ACPF

This method is represented by an object of type *ACPF* and solves an AC power flow problem. For doing this, it can use the NR solver from OPTALG together with "switching" heuristics for modeling local controls. Alternatively, it can formulate the problem as an optimization problem with a convex objective function and *complementarity constraints*, *e.g.*, Voltage set-point regulation by generators, for modeling local controls, and solve it using the INLP, AugL, or Ipopt solver available through OPTALG.

The parameters of this power flow method are the following:

Name	Description	Default
`'weight_vmag'`	Weight for bus voltage magnitude regularization	`1e0`
`'weight_vang'`	Weight for bus voltage angle regularization	`1e-3`
`'weight_powers'`	Weight for generator power regularization	`1e-3`
`'weight_var'`	Weight for generic regularization	`1e-5`
`'Q_limits'`	Flag for enforcing generator Q limits	`True`
`'Q_mode'`	Reactive power mode (free or regulating)	`regulating`
`'pvpq_start_k'`	Start iteration number for PVPQ switching heuristics	`0`
`'vmin_thresh'`	Low-voltage threshold	`1e-1`
`'solver'`	OPTALG optimization solver `{'nr','inlp','augl','ipopt'}`	`'nr'`

2.4 ACOPF

This method is represented by an object of type *ACOPF* and solves an AC optimal power flow problem. For doing this, it uses the INLP, AugL, or Ipopt solver from OPTALG. By default, it minimizes Active power generation cost subject to voltage magnitude limits, generator power limits, *e.g.*, Variable bounds, and AC Power balance.

The parameters of this optimal power flow method are the following:

Name	Description	Default
`'weight_cost'`	Weight for active power generation cost	`1e0`
`'weight_vmag'`	Weight for bus voltage magnitude regularization	`0.`
`'weight_vang'`	Weight for bus voltage angle regularization	`0.`
`'weight_pq'`	Weight for generator power regularization	`0`
`'thermal_limits'`	Branch thermal limits `{'none','linear','nonlinear'}`	`'none'`
`'vmin_thresh'`	Low-voltage threshold	`1e-1`
`'solver'`	OPTALG optimization solver `{'augl','inlp','ipopt'}`	`'inlp'`

CHAPTER 3

Command-Line Utility

This section provides details about how to use the `gridopt` command-line utility.

3.1 Syntax

The command-line utility `gridopt` can be used to load a network data file and solve the corresponding PF or OPF problem using an available method. One can also configure the parameters of the underlying optimization solver as well as the properties of the optimization problem constructed. The syntax and a description of each of the options is presented below.

Power grid optimization package.

```
usage: gridopt [-h] [--parameters [PARAMS [PARAMS ...]]] [--profile]
               [--flatstart] [--quiet] [--write OUTFILE]
               case method
```

3.1.1 Positional Arguments

case filename of power flow case to solve

method Possible choices: ACOPF, ACPF, DCOPF, DCPF

PF or OPF method

3.1.2 Named Arguments

--parameters parameter name-value pairs

Default: []

--profile flag for profiling execution

Default: False

--flatstart	flag for flat starting point
	Default: False
--quiet	flag for supressing output
	Default: False
--write	name of output file
	Default: “”

3.2 Example

The following example shows how to use the command-line utility to solve an AC power flow problem using the Newton-Raphson algorithm with a feasibility tolerance of `1e-5` per unit system MVA:

```
gridopt ieee14.m ACPF --parameters feastol=1e-5 solver=nr
```

In this example, it is assumed that the command is executed from a directory where the sample case ieee14 is located.

CHAPTER 4

API Reference

4.1 Power Flow Method

gridopt.power_flow.**new_method**(*name*)
Creates a power flow or optimal power flow method.

Parameters

name [{'DCPF', 'DCOPF', 'ACPF', 'ACOPF'}]

class gridopt.power_flow.method.**PFmethod**
Power flow method class.

create_problem(*self*, *net*)
Creates optimization problem.

Parameters

net [Network]

Returns

prob [Problem]

get_info_printer(*self*)
Gets function for printing information about method progress.

Returns

printer [Function]

get_parameters(*self*)
Gets method parameters.

Returns

params [dict]

get_results(*self*)
Gets dictionary with results.

Returns

results [dict]

set_network_snapshot (*self*, *net*)
Sets network snapshot.

Parameters

net [Network]

set_parameters (*self*, *params=None*, *strparams=None*)
Sets method parameters.

Parameters

params [dict] Name-value pairs

strparams: dict Name-value pairs where value is a string

set_problem (*self*, *p*)
Sets problem.

Parameters

p [Problem]

set_problem_time (*self*, *t*)
Sets problem construction time in seconds.

Parameters

t [float]

set_results (*self*, *results*)
Sets method results.

Parameters

results [dict]

set_solver_dual_variables (*self*, *d*)
Sets solver dual variables.

Parameters

d [list]

set_solver_iterations (*self*, *k*)
Sets solver iterations.

Parameters

k [int]

set_solver_message (*self*, *msg*)
Sets solver message.

Parameters

msg [string]

set_solver_name (*self*, *name*)
Sets solver name.

Parameters

name [string]

set_solver_primal_variables(*self*, *x*)
Sets solver primal variables.

Parameters

x [vector]

set_solver_status(*self*, *status*)
Sets solver status.

Parameters

status [string]

set_solver_time(*self*, *t*)
Sets solver time in seconds.

Parameters

t [float]

solve(*self*, *net*)
Solves power flow problem.

Parameters

net [Network]

update_network(*self*, *net*)
Updates network with results.

Parameters

net [Network]

class gridopt.power_flow.dc_pf.**DCPF**
DC power flow method.

class gridopt.power_flow.dc_opf.**DCOPF**
DC optimal power flow method.

class gridopt.power_flow.ac_pf.**ACPF**
AC power flow method.

class gridopt.power_flow.ac_opf.**ACOPF**
AC optimal power flow method.

4.1.1 Error Exceptions

class gridopt.power_flow.method_error.**PFmethodError**

class gridopt.power_flow.method_error.**PFmethodError_NoProblem**

class gridopt.power_flow.method_error.**PFmethodError_BadProblem**

class gridopt.power_flow.method_error.**PFmethodError_BadFlowLimits**

class gridopt.power_flow.method_error.**PFmethodError_BadVarLimits**

class gridopt.power_flow.method_error.**PFmethodError_BadParams**(*keys*)

class gridopt.power_flow.method_error.**PFmethodError_BadOptSolver**(*param=''*)

class gridopt.power_flow.method_error.**PFmethodError_ParamNotBool**

class gridopt.power_flow.method_error.**PFmethodError_SolverError**(*msg*)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

U